Universidad Autónoma de Madrid
Escuela Politécnica Superior

Grado en Ingeniería de Informática

Trabajo Fin de Grado

REDES RECURRENTES PARA EL ANÁLISIS DE SERIES TEMPORALES

Autor: Javier Román Morales
Tutor: Luis Fernando Lago Fernández

Julio 2018

REDES RECURRENTES PARA EL ANÁLISIS DE SERIES TEMPORALES

AUTOR: Javier Román Morales
TUTOR: Luis Fernando Lago Fernández

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2018

Resumen

Resumen

Las redes neuronales recurrentes se han convertido hoy por hoy en una de las herramientas más potentes del aprendizaje automático y del modelado de secuencias. En los últimos años estas redes han conseguido fantásticos resultados y cada vez las tenemos más presentes en nuestro día a día, por ejemplo, el traductor de Google o sistemas de procesamiento de lenguaje natural usan redes neuronales de este tipo.

En este trabajo de fin de grado se llevará a cabo una investigación sobre redes neuronales recurrentes. En primer lugar se hará una introducción a los conceptos básicos de redes neuronales para pasar a explicar más a fondo el funcionamiento de modelos recurrentes como la LSTM o la GRU. Mencionaremos trabajos llevados a cabo con modelos recurrentes y para finalizar realizaremos una introducción a algunas de las herramientas más importantes usadas en este campo del aprendizaje automático.

Diseñaremos, con este tipo de redes, experimentos que nos permitan comprender que modelo recurrente da mejores resultados y el ganador lo intentaremos aplicar a la generación de texto y música. Para esto, explicaremos los datos usados para el entrenamiento y el procedimiento llevado a cabo sobre estos en el caso de que necesiten un preprocesamiento. Se aplicarán métricas para evaluar como de buenos son los resultados musicales generados.

Palabras Clave

Aprendizaje automático, redes neuronales, LSTM, generación de música, generación de texto.

Abstract

Recurrent neural networks have become one of the most powerful tools in machine learning nowadays. In the last few years this kind of networks have achieved great results as they have become a present part of our lives, for instance, Google translator or natural language processing systems use this kind of networks.

In this thesis we will investigate about recurrent neural networks. First we will introduce basic concepts about neural networks and after that we will explain how some recurrent models as LSTM or GRU work. We will mention other work with these networks and to finish we will introduce some of the most important tools used in this field of machine learning.

We will design experiments based on those recurrent models to be able to understand which model gets the best result and this model will be applied to text and music generation. We will explain the chosen datasets for training our models and the procedure made over that data if needed. Musical results will be evaluated to know how good they are.

Key words

Machine learning, neural networks, LSTM, music generation, text generation.

Agradecimientos

Al colegio Sagrados Corazones de Martín de los Heros y a sus profesores por darme toda la educación necesaria con la que pude llegar a la universidad y por hacerme interesarme por la informática y la tecnología por primera vez.

A la Escuela Politécnica Superior y la Universidad Autónoma por ofrecer un ambiente acogedor y una educación actualizada de calidad.

A todos los profesores que a lo largo de la carrera me han enseñado tanto y en especial a Luis Lago, mi tutor, que me abrió puertas a mundos como las redes neuronales recurrentes o los concursos de programación rápida que luego tanto me han interesado.

A mis amigos, en especial a los de clase, alias Yatekomo, que tanta ayuda me han dado y con los que tantos momentos maravillosos he pasado en estos cuatro años, y con los que muchos más momentos así espero seguir pasando.

A mis tíos y primos con los que siempre he podido compartir mi vocación por la informática y recibir a cambio su vocación por la medicina.

A mis abuelos, quienes han sido uno de los pilares más importantes de mi vida hasta ahora y una gran inspiración por avanzar.

A mi hermano, por ser un amigo con quien hablar y reírme y con quien compartir mis aficiones.

Y para finalizar, a mis padres, quienes me han educado y ofrecido siempre todas las oportunidades y el amor del mundo y lo más importante, quienes me han dado la vida.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

1.1. Motivación

En la actualidad, las redes neuronales profundas están tomando el protagonismo dentro del campo del aprendizaje automático. Grandes empresas tecnológicas como Google, Amazon o Facebook están investigando y aplicando estas tecnologías a diversos campos.

Uno de los campos de aplicación es el modelado de secuencias. Este es un campo interesante sobre el que aplicar aprendizaje automático ya que numerosos procesos que encontramos en nuestro día a día forman series temporales.

Tradicionalmente para el análisis de secuencias se han aplicado modelos probabilistas tales como el modelado por cadenas de Markov [5]. Hoy en día debido al auge de las Redes Neuronales Artificiales se comienzan a aplicar modelos basados en esta tecnología.

Dentro del campo de las Redes Neuronales Artificiales encontramos diferentes familias, como las *Feedforward*, que engloban todas aquellas arquitecturas de red en donde las conexiones entre las neuronas no forman un ciclo. Estas no dan buen resultado para el modelado de secuencias debido a que tendrían parámetros separados para cada una de las entradas por lo que aprenderían las reglas de nuestra secuencia temporal por separado [3].

Sin embargo las Redes Neuronales Recurrentes (RNNs por sus siglas en inglés) [3] son redes en cuyos esquemas las conexiones entre cada una de las unidades forman un grafo dirigido a lo largo de una secuencia. Esto permite que la red comparta los mismos pesos a lo largo de varios pasos de tiempo.

En este TFG realizaremos una investigación sobre distintos tipos de Redes Neuronales Recurrentes y las aplicaremos a labores de generación y modelado de series temporales.

1.2. Objetivos

El objetivo de este trabajo de fin de grado es el de realizar un estudio del estado del arte en relación al modelado de secuencias mediante Redes Neuronales Recurrentes, comprender su funcionamiento y experimentar con distintos tipos de RNNs sobre series temporales sencillas.

Estas redes dan muy buenos resultados para datos con secuencias temporales intrínsecas como por ejemplo sonido, lenguaje, traducción, conversaciones o valores bursátiles.

Dado que tanto la música como el texto son secuencias temporales, realizaremos pruebas encaminadas a la generación de música y texto.

1.3. Estructura del documento

Este documento se estructura en un total de seis capítulos. El capítulo de **estado del arte** realiza una introducción al modelado de secuencias así como a las redes neuronales, desarrolla distintos modelos de redes recurrentes y herramientas usadas en aprendizaje automático.

En el apartado de **diseño** se introducen los experimentos a llevar a cabo, se explica tanto el planteamiento para el desarrollo de los mismos así como los objetivos perseguidos en cada uno de ellos.

El capítulo de **desarrollo** expone todas y cada una de las actividades llevadas a cabo en la investigación sobre las redes recurrentes. Se enumeran también detalles técnicos de cada experimento, como el tipo de modelo a implementar o las operaciones de preprocesamiento a realizar sobre los datos para cada experimento.

Los **resultados** se muestran en el capítulo con el mismo nombre. En este se exhiben cada uno de los resultados de las pruebas realizadas y se comenta su significado.

Cerramos el documento con las **conclusiones** en donde se realizará una reflexión sobre las redes recurrentes y la información sacada en claro con nuestra investigación.

En el apartado de **anexos** se adjunta el código de algunos de los modelos probados.

2

Estado del arte

2.1. Introducción

En este apartado, se realiza una pequeña introducción a conceptos básicos de redes neuronales para posteriormente pasar a comentar en más profundidad las redes recurrentes así como sus subtipos. Finalmente se hablará de las herramientas *TensorFlow* y *Keras* y su funcionamiento.

2.2. Conceptos básicos de las redes neuronales.

2.2.1. La Red.

Las redes neuronales artificiales, basan su funcionamiento en la conexión de unidades básicas llamadas neuronas.

Definición 2.2.1. *Una **neurona artificial** [6] es una función matemática que se inspira en los modelos biológicos. Estas realizan una suma ponderada de sus entradas, representando así la combinación de señales inhibidoras y excitadoras que cada neurona biológica realiza, y el resultado lo pasan a través de una función de activación. Esto representa el potencial de acción con el que la celula dispara. (Figura 2.1)*

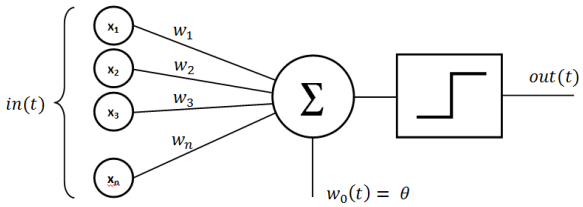

Figura 2.1: Esquema de funcionamiento de una neurona artificial. Este muestra las entradas a la neurona y los pesos de las conexiones a la misma. Además indica que se realiza la suma pesada de entradas y pesos y que el resultado se pasa a través de una función de activación no lineal. Imagen de `https://www.safaribooksonline.com/library/view/machine-learning-for/9781491971444/ch01.html`.

La **función de activación** de la neurona suele elegirse de tal modo que introduzca no linealidad en el sistema. Normalmente esta función, también conocida como **función de transferencia**, suele ser monótona creciente, continua, diferenciable y acotada.

Cada una de estas neuronas se conecta con el resto de neuronas a través de un enlace con un peso asociado. La neurona aprende ajustando estos pesos durante el proceso de entrenamiento.

Las arquitectura de red más típica y representativa es la *feedforward* también conocida como **perceptrón multicapa**. Se muestra en la figura 2.2. Esta red organiza todas las neuronas en capas. Estas capas conectan sus neuronas con todas las neuronas de la siguiente capa.

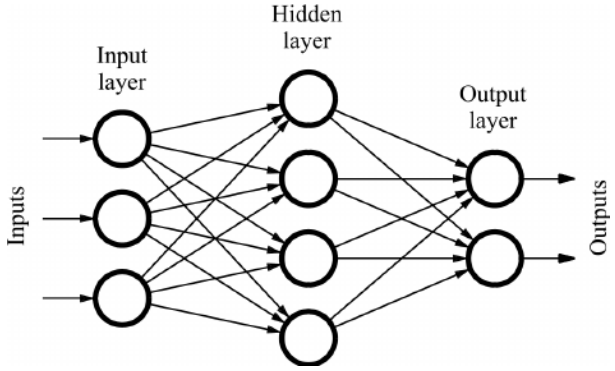

Figura 2.2: Perceptrón multicapa. En esta se muestran tres capas, una capa de entrada con tres neuronas, una capa intermedia u oculta con cuatro neuronas y una capa de salida. Entre todo par de neuronas en capas adyacentes existe una conexión. Imagen de `https://medium.com/bbm406f17/week-3-like-i-like-1252958e3bf0`

Hoy en día se construyen redes que se catalogan dentro de la categoría de *redes profundas*. Estas redes encadenan gran cantidad de capas ocultas permitiendo así el modelado de relaciones no lineales complejas al poder abstraer características de más alto nivel de los datos de entrada [7].

Definición 2.2.2. *La **propagación hacia delante** es el proceso en el que una red calcula su salida a partir de sus entradas. Esto lo lleva a cabo aplicando operaciones matemáticas sobre las entradas de cada capa para obtener la salida de esa misma capa. Esto se realiza hasta hallar el resultado de la última capa. [6]*

Si llamamos $\boldsymbol{y}$ al vector de salidas, $\boldsymbol{x}$ al vector de entradas y W a la matriz de pesos de una capa, la operación básica que se realiza en una **propagación hacia delante** en cualquier capa quedaría como sigue:

$$\mathbf{y} = f(\mathbf{x} \cdot W) \tag{2.1}$$

En esta ecuación, el símbolo f indica la función de activación de las neuronas de la capa. Esta se aplica para todas las neuronas obteniendo así un vector de salidas de la capa que se usará como entrada a la siguiente.

Como antes mencionamos, cada neurona de la red tiene su función de activación y normalmente todas las neuronas se suelen disponer con la misma función de activación, aunque esto no tiene por que ocurrir siempre. Dos de las funciones más conocidas son la **sigmoide** y la **tangente hiperbólica**. Ver figura 2.3.

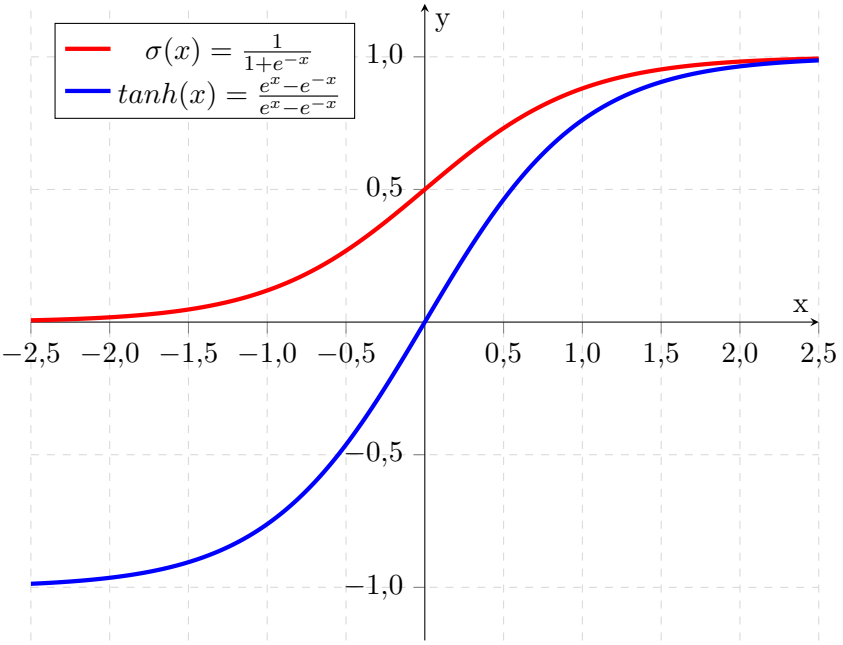

Figura 2.3: Función sigmoide y tangente hiperbólica.[1]

Una de las ventajas de estas dos funciones mostradas en la figura 2.3 es que su derivada se puede expresar en términos de ella misma. Aún así cuentan con un problema, que sobre todo aparece en redes que encadenan muchas capas. Este se llama el problema de los *vanishing gradients*[8].

Aparece debido a la naturaleza de estas funciones, para valores muy negativos o positivos de la x, la curva se asemeja mucho a una linea horizontal por lo que la derivada en estos puntos será muy cercana a cero. Debido a que en retropropagación se utiliza la regla de la cadena para calcular los gradientes, estos valores cercanos a cero de la derivada se irán acumulando multiplicativamente haciendo que el gradiente se haga cada vez más y más cercano a cero, sobre todo en las primeras capas de la red. Debido a que la precisión de los ordenadores actuales no es infinita, muchas veces un valor tan cercano a cero, acabará siendo representado como cero en un ordenador, haciendo que el gradiente desaparezca por completo.

Tener gradientes de cero es malo debido a que la red puede dejar de aprender o no aprender correctamente.

Otras funciones ampliamente usadas, sobre todo en las redes profundas son funciones tales como la *ReLU* y la *Leaky ReLU*[1]. Se muestran en la figura 2.4.

Estas funciones de activación tienen ventajas sobre la sigmoide, por ejemplo, reducen la probabilidad de sufrir el problema de los *vanishing gradients* y su cálculo es más rápido al no involucrar operaciones exponenciales. Además, el gradiente de estas funciones es siempre un valor constante lo que produce un aprendizaje más rápido.

Estas funciones se suelen usar en capas ocultas, no son válidas para la última capa ya que no se les puede dar la interpretación probabilista que si se le puede dar a los resultados de la sigmoide.

En general, la mayoría de redes neuronales suelen formar parte de un paradigma de aprendizaje conocido como ***aprendizaje supervisado***[6]. En este los patrones de entrenamiento cuentan con una clase objetivo. La red compara la predicción que da para un determinado patrón con la clase real de ese patrón y aprende en base al error que comete. Esto se explica más a fondo en la sección 2.2.2.

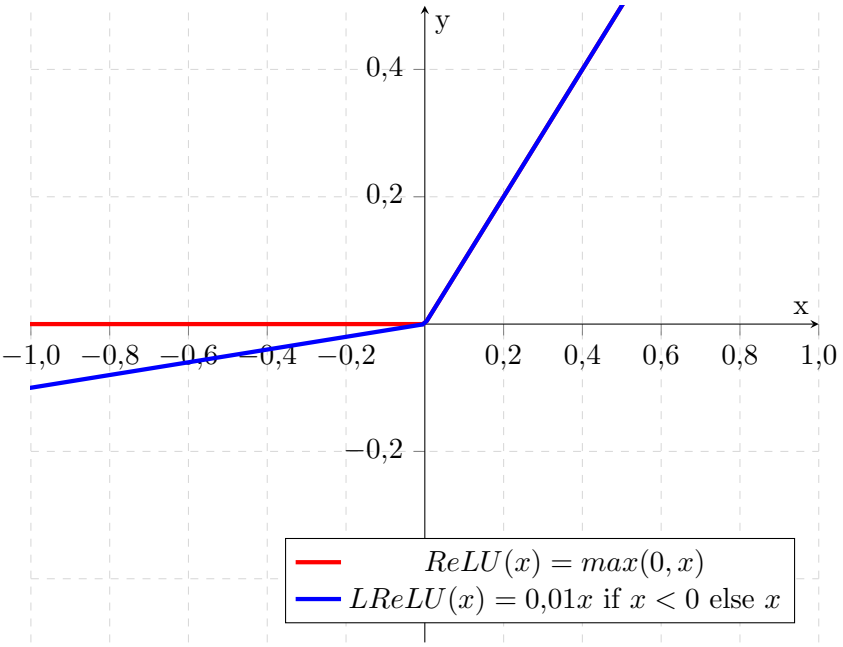

Figura 2.4: Función *ReLU* y *Leaky ReLU*

2.2.2. El algoritmo de retropropagación.

Como anteriormente comentamos, las redes neuronales se forman por la conexión de neuronas organizadas en distintas capas. La red aprende actualizando los pesos de las conexiones entre sus neuronas con lo observado con cada patrón de entrenamiento. El proceso mediante el cual actualiza estos pesos es **el algoritmo de retropropagación**[6].

Este intenta minimizar la función de error en el espacio de los pesos. Esto lo realiza utilizando el método del **descenso de gradiente**[3]. Las combinaciones de pesos que más minimizan el error se consideran soluciones buenas. Aún así, debido a que la función de error puede tener varios mínimos también puede haber varias soluciones.

Definición 2.2.3. *El **gradiente** de una función matemática, denotado como ∇f es un vector que, evaluado en un punto del espacio nos indicará la dirección de más rápido crecimiento de la función desde ese punto. En este vector las coordenadas serán las derivadas parciales de la función en el punto que está siendo evaluado.*

$$\nabla f(p) = \left(\frac{\partial f(p)}{\partial x_1} \frac{\partial f(p)}{\partial x_2}, ..., \frac{\partial f(p)}{\partial x_n} \right) \tag{2.2}$$

Definición 2.2.4. *El **descenso de gradiente** es un algoritmo de optimización iterativo que tomando pasos en la dirección negativa del **gradiente** logra encontrar mínimos locales de una determinada función.*

Como ya hemos explicado con anterioridad, el gradiente de una función indica la dirección de máximo crecimiento de la misma. Por lo tanto los pesos de la red se actualizarán en la dirección opuesta a este para así ir minimizando el error.

Para poder explicar como funciona el algoritmo de retropropagación matemáticamente[9][10][6] tenemos que definir cierta notación:

- w_{ij}^k: Es el peso que conecta el elemento i de la capa $k-1$ con el elemento j de la capa k.

- b_j^k: Es el sesgo (o *bias*) del nodo j de la capa l_k.
- a_i^k: Es el resultado de la suma pesada de las entradas a la capa con sus respectivos pesos para el elemento i de la capa l_k.
- o_i^k: Salida del elemento i de la capa l_k.
- r_k: Número de elementos en la capa l_k.
- $\hat{y}$: Salida de la red.
- y: Clase esperada para un cierto patrón.
- $g(x)$: Función de activación.
- $g'(x)$: Derivada de la función de activación.

El primer paso sería propagar hacia delante en la red por cada patrón de entrenamiento para obtener una predicción. Una vez tenemos esta predicción trataríamos de minimizar el error al que llamaremos con la letra E.

Para minimizar el error se deriva esta función de error con respecto a cada uno de los pesos de la red. Para facilitar la notación de las ecuaciones asumiremos lo siguiente:

$$b_j^k = w_{0j}^k \tag{2.3}$$

Esto quiere decir que el sesgo de cualquiera de los elementos j de la capa k se corresponderá con el peso 0 de cada elemento. Por lo tanto el valor de activación de la neurona i de la capa k vendrá determinado por la fórmula siguiente:

$$a_j^k = \sum_{i=0}^{r_{k-1}} w_{ij}^k o_i^{k-1} \tag{2.4}$$

En este caso o_i^{k-1} indica las salidas de las neuronas de la capa anterior que conectan con la neurona que estemos evaluando, esto es, las entradas a la neurona evaluada. Se añadirá un 1 como valor de entrada para el sesgo de la neurona. Es decir, $o_0^{k-1} = 1$.

Conociendo el resultante a_j^k de la suma pesada de las entradas a la neurona podemos calcular el valor de salida de la misma pasando este valor a través de la función de activación que estemos utilizando:

$$o_j^k = g(a_j^k) \tag{2.5}$$

Si nuestra red cuenta con un total de L capas las salidas de la última capa se corresponderán con las salidas de la red:

$$\hat{y}_j = o_j^L \tag{2.6}$$

Las cuatro fórmulas anteriores nos permiten realizar el proceso de propagación hacia delante en la red.

Con el resultado de salida, para empezar, calculamos el error de $\hat{y}$ con respecto de y y hallamos el gradiente de este con respecto a cada uno de los pesos de la última capa (L) aplicando la regla de la cadena:

$$\frac{\partial E}{\partial w_{ij}^L} = \frac{\partial E}{\partial a_j^L}\frac{\partial a_j^L}{\partial w_{ij}^L} = \frac{\partial E}{\partial o_j^L}\frac{\partial o_j^L}{\partial a_j^L}\frac{\partial a_j^L}{\partial w_{ij}^L} \tag{2.7}$$

El término $\frac{\partial E}{\partial a_j^L}$ de la ecuación anterior es la **señal de error** del elemento j de la última capa. Representa cuanto cambia el error cuando la suma pesada de las entradas a la neurona cambia. Por notación lo llamaremos δ_j^L:

$$\delta_j^L \equiv \frac{\partial E}{\partial a_j^L} = \frac{\partial E}{\partial o_j^L}\frac{\partial o_j^L}{\partial a_j^L} = \frac{\partial E}{\partial o_j^L}g'(a_j^L) \tag{2.8}$$

La señal de error de la última capa depende por lo tanto de hallar la derivada del error entre nuestra predicción y el valor objetivo con respecto de las salidas de esta capa (vector o). Esta dependerá de la función de error seleccionada.

Si derivamos el término $\frac{\partial a_j^L}{\partial w_{ij}^L}$ de la ecuación 2.7 obtenemos:

$$\frac{\partial a_j^L}{\partial w_{ij}^L} = \frac{\partial}{\partial w_{ij}^L}\sum_{l=0}^{r_{L-1}} w_{lj}^L o_i^{L-1} = o_i^{L-1} \tag{2.9}$$

Por lo tanto la ecuación 2.7 la podemos escribir como:

$$\frac{\partial E}{\partial w_{ij}^L} = \left(\frac{\partial E}{\partial o_j^L}g'(a_j^L)\right)o_i^{L-1} = \delta_j^L o_i^{L-1} \tag{2.10}$$

Con esto obtenemos el gradiente del error con respecto de los pesos de la última capa. Algo interesante ahora es calcular el gradiente para las capas ocultas. En este caso la capa evaluada siempre cumplirá $1 \leq k < L$. Para esto derivamos el error con respecto de los pesos:

$$\begin{gathered}\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial E}{\partial o_j^k}\frac{\partial o_j^k}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} = \\ \sum_{l=1}^{r_{k+1}}\frac{\partial E}{\partial o_l^{k+1}}\frac{\partial o_l^{k+1}}{\partial a_l^{k+1}}\frac{\partial a_l^{k+1}}{\partial o_j^k}\frac{\partial o_j^k}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k}\end{gathered} \tag{2.11}$$

Sacando factor común $\frac{\partial o_j^k}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k}$ al sumatorio mostrado en la ecuación anterior obtenemos:

$$\begin{gathered}\frac{\partial E}{\partial w_{ij}^k} = \left(\sum_{l=1}^{r_{k+1}}\frac{\partial E}{\partial a_l^{k+1}}\frac{\partial a_l^{k+1}}{\partial o_j^k}\right)\frac{\partial o_j^k}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} = \\ \left(\sum_{l=1}^{r_{k+1}}\frac{\partial E}{\partial a_l^{k+1}}w_{jl}^{k+1}\right)g'(a_j^k)o_i^{k-1}\end{gathered} \tag{2.12}$$

El sumatorio anterior se debe al hecho de que cada neurona j de una capa k asociada con el peso w_{ij}^k sobre el que estamos derivando depende del error asociado con todas las neuronas de la capa siguiente r_{k+1}. Esto se puede observar en la figura 2.2.

A continuación, sería interesante generalizar la definición de **señal de error** para cualquier capa. Al igual que en 2.8 podemos decir:

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k} \tag{2.13}$$

En la ecuación 2.11 ya hemos expandido el término $\frac{\partial E}{\partial a_j^k}$ de tal manera que nos quedaría lo siguiente:

$$\frac{\partial E}{\partial a_j^k} = \left(\sum_{l=1}^{r_{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial o_j^k} \right) \frac{\partial o_j^k}{\partial a_j^k} = \left(\sum_{l=1}^{r_{k+1}} \frac{\partial E}{\partial a_l^{k+1}} w_{jl}^{k+1} \right) g'(a_j^k) \tag{2.14}$$

Aplicando 2.13 sobre 2.14 obtenemos una definición recursiva de δ_j^k:

$$\delta_j^k = \left(\sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \right) g'(a_j^k) \tag{2.15}$$

Con esta definición de la señal de error, la ecuación de la derivada del error con respecto de un peso de la capa oculta quedaría:

$$\frac{\partial E}{\partial w_{ij}^k} = \left(\sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \right) g'(a_j^k) o_j^{k-1} = \delta_j^k o_i^{k-1} \tag{2.16}$$

Una vez conocemos las bases matemáticas del algoritmo, únicamente deberíamos actualizar los pesos en la dirección contraria al gradiente:

$$\Delta w_{ij}^k = -\alpha \frac{\partial E}{\partial w_{ij}^k} \tag{2.17}$$

El alfa por el que se multiplica en la función anterior es la constante de aprendizaje. Este es un valor, normalmente en el intervalo $(0, 1]$, que le indica a la red cómo de grandes han de ser los ajustes realizados a los pesos.

2.3. Redes recurrentes

2.3.1. Introducción.

Las Redes Neuronales Recurrentes (RNNs por sus siglas en inglés) son una familia de redes neuronales muy útiles para el procesamiento de secuencias. Estas redes pueden escalar secuencias mucho más largas que las que podría generalizar cualquier red no especializada en este aspecto. Además, pueden procesar secuencias de longitud variable.

Una de las ideas principales que propició el desarrollo de estas redes fue extraída de las técnicas de aprendizaje automático y de los modelos estadísticos encontrados en la década de los 80. Esta idea se basaba en compartir diferentes parámetros a lo largo de diferentes partes de un modelo. Esto permitió generalizar conjuntos de datos que contaban con dependencias temporales entre sus entradas [3].

Un perceptrón multicapa tendría muy difícil el hecho de generalizar sobre una secuencia debido a que, si entrenamos esta red con secuencias de longitud fija, la red tendría parámetros separados para cada una de las características de entrada teniendo que aprender todas las

reglas por separado. Sin embargo, la RNNs encontraría esta tarea mucho más fácil debido a que comparte pesos a lo largo del tiempo. Cabe mencionar que cuando mencionamos el tiempo no tiene por que significar literalmente tiempo, puede simplemente significar la posición de un objeto en una secuencia.

Como podemos ver en el paradigma tradicional de las redes neuronales se toman las características de la entrada como independientes entre ellas. Sin embargo en las redes recurrentes la salida de un nodo depende de las salidas de los nodos anteriores de la red.

Una RNN cuenta con una recursión o recurrencia debido a que este tipo de redes realizan la misma operación sobre todos los elementos de la secuencia compartiendo información obtenida del elemento anterior. Puede ser representada de una manera compacta como se muestra a continuación.

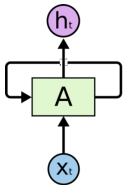

Figura 2.5: Representación básica y compacta de una red recurrente[2].

Normalmente la red se suele desdoblar en una secuencia finita, intentando predecir el siguiente elemento de la secuencia mirando unicamente N pasos atrás, de manera que sea viable entrenar y obtener resultados de esta [11].

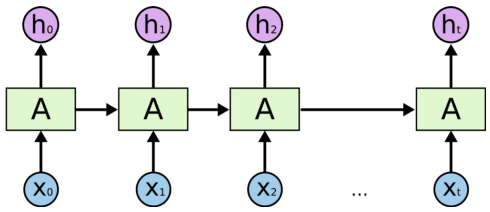

Figura 2.6: Representación básica de una red recurrente desdoblada[2].

2.3.2. RNN Básica

Pese a que hay distintos patrones de diseño, una de las redes recurrentes más simples que podemos construir es aquella que produce una salida por cada paso de tiempo y con conexiones entre las unidades ocultas. A este red lo llamaremos también *Vainilla RNN*. Fueron mencionadas por primera vez en 1990 por Jeffrey L. Elman [12].

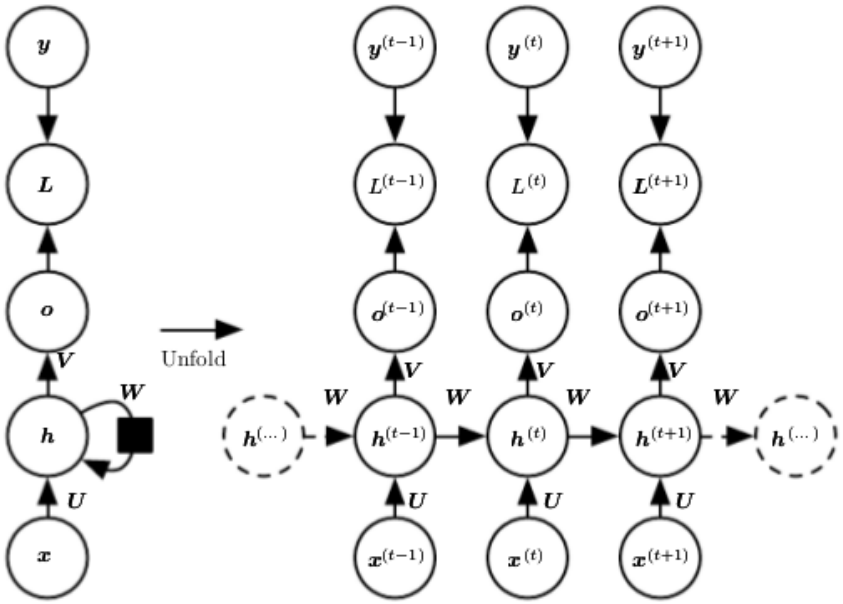

Figura 2.7: Representación de una Vainilla RNN. [3]

Esta red se puede formalizar matemáticamente de la siguiente manera

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}, \tag{2.18}$$

$$\mathbf{h}^{(t)} = tanh(\mathbf{a}^{(t)}), \tag{2.19}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}, \tag{2.20}$$

$$\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)}) \tag{2.21}$$

En esta fórmula los parámetros b y c representan los vectores del *bias* y los parámetros U, V y W representan las matrices de pesos para las conexiones entre entrada y capa-oculta, capa-oculta y salida y capa-oculta y capa-oculta respectivamente. Además, a la salida de **o** le aplicamos la función *softmax*, esta nos hará una transformación del vector resultante a un vector de probabilidades, asegurándonos que la suma de todos los valores de como resultado 1.

$$softmax(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \; para \; k = 1, ..., K \tag{2.22}$$

La L que se muestra en la figura 2.7 indica el *loss* (error) de la predicción con respecto del objetivo **y**. En este caso, el *loss* total de la secuencia sería la suma de todos los *losses* a lo largo de toda la secuencia.

Una vez se propaga la información hacia delante, debemos hacer retropropagación en nuestra red recurrente. Pero dado que cada nodo depende de los nodos anteriores, el gradiente de un nodo ha de ser propagado también hacia atrás, es decir, debemos retropropagar en el tiempo. A este algoritmo se le conoce como Retropropagación en el Tiempo o *BPTT* (*Back-propagation Through Time*).

Aunque esta red recurrente puede llegar a aprender, tiene ciertos problemas. Dado que encadenan numerosos nodos, el gráfo se hace muy profundo y suele dar problemas de *vanishing* o *exploding gradients*.

Una solución que se le suele dar es el ***gradient clipping***. Esta técnica evita que el gradiente explote o desaparezca reescalandolo para que su norma se encuentre como máximo en un valor determinado.

2.3.3. LSTM.

Los modelos que mejor resultados dan para el modelado de secuencias son todos aquellos enmarcados dentro de la categoría de *Gated RNNs*. La idea de este tipo de redes es la de crear caminos en el tiempo que no produzcan problemas de *vanishing* o *exploding gradients*. Además, permiten acumular información durante mucho tiempo e incluso pueden olvidar aquella que para la red ya no sea útil.

Las redes LSTM [13][3] entran en la familia de las *Gated RNNs* y dan muy buenos resultados a la hora de aprender dependencias temporales largas. Fueron introducidas por Hochreiter y Schmidhuber en el año 1997 [14] posteriormente fueron mejoradas por Gers en el año 2000 [15].

Las unidades LSTM (Long Short-Term Memory) son unidades usadas para la construcción de redes recurrentes. Cada unidad LSTM es una **celda** con cuatro puertas (gates) la **input-gate**, la **external-input-gate**, la **forget-gate** y **output-gate**. Lo más importante de la celda es su **estado interno** encargado de recordar valores a lo largo del tiempo.

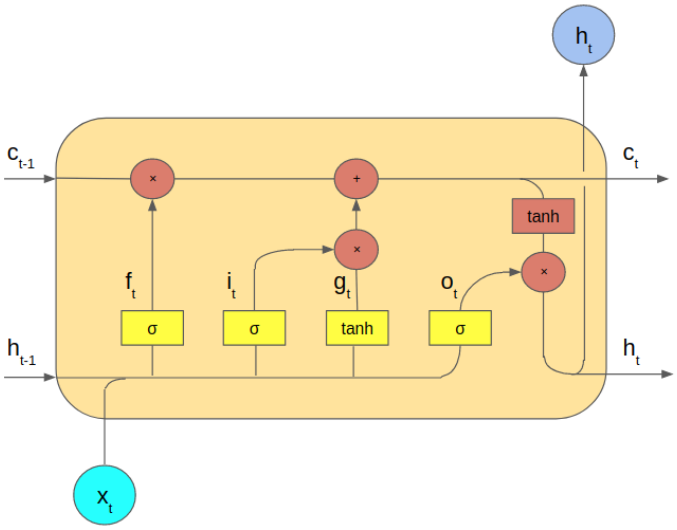

Figura 2.8: Representación de una celda LSTM.

El **estado interno** viene representado en el diagrama anterior por C, este recorre toda las celdas encadenadas sufriendo pequeños cambios cada vez que pasa por cada una de ellas.

El ***hidden-state*** es sin embargo, la salida de cada una de las unidades. Este también se comparte a lo largo de todas las unidades de nuestro modelo. Está representado con la letra h.

El primer cambio que el **estado interno** sufre es el provocado por la ***forget-gate*** (f_t). Esta decide si el valor que tiene el estado de la celda ha de ser olvidado o no. Esto se realiza aplicando sobre la entrada y el *hidden-state* de la celda anterior la función sigmoide y multiplicando el resultado por el **estado interno** que viene de la celda anterior. Dado que esta función toma valores entre 0 y 1, podrá dar de resultado 0 y provocar que se olvide, o dar como resultado 1 significando que el valor anterior ha de ser completamente recordado.

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{b}^f + U^f \boldsymbol{x}^{(t)} + W^f \boldsymbol{h}^{(t-1)}\right) \tag{2.23}$$

Una vez se ha decidido si olvidar o recordar lo que teníamos en el **estado interno** procederemos a decidir que nuevos valores guardar en este. Esto se hace mediante una combinación

entre la ***input-gate*** (i_t) y la ***external-input-gate*** (g_t). La primera decide los valores a actualizar y la segunda crea un vector de nuevos candidatos. La combinación de ambas crea una actualización para el **estado interno**.

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{b}^i + \boldsymbol{U}^i\boldsymbol{x}^{(t)} + \boldsymbol{W}^i\boldsymbol{h}^{(t-1)}\right), \tag{2.24}$$

$$\boldsymbol{g}^{(t)} = \tanh\left(\boldsymbol{b}^g + \boldsymbol{U}^g\boldsymbol{x}^{(t)} + \boldsymbol{W}^g\boldsymbol{h}^{(t-1)}\right) \tag{2.25}$$

La actualización de nuestro estado vendría dado por lo siguiente:

$$\boldsymbol{C}^{(t)} = \boldsymbol{f}^{(t)}\boldsymbol{C}^{(t-1)} + \boldsymbol{i}^{(t)}\boldsymbol{g}^{(t)} \tag{2.26}$$

Una vez tenemos el nuevo **estado interno** de la celda debemos calcular cual será la nueva salida $h^{(t)}$ de nuestra celda.

En primer lugar mediante la **output-gate** ($o^{(t)}$) seleccionamos los valores del **estado interno** que devolveremos, con los resultados obtenidos tras aplicar la sigmoide a la combinación de las entradas $x^{(t)}$ y la salida anterior $h^{(t-1)}$.

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{b}^o + U^o\boldsymbol{x}^{(t)} + W^o\boldsymbol{h}^{(t-1)}\right) \tag{2.27}$$

Tras esto pasamos los valores del **estado interno** a través de una *tanh* y lo combinamos con la salida de la **output-gate** obteniendo nuestro nuevo $h^{(t)}$.

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)}tanh(\boldsymbol{C}^{(t)}) \tag{2.28}$$

2.3.4. GRU

Las GRUs (Gate Recurrent Unit)[16][3] por sus siglas en inglés surgen frente a la necesidad de reducir los tiempos de entrenamiento de la LSTM sin perder el desempeño que esta tiene.

Las GRUs responden a la pregunta de qué elementos de la arquitectura de la LSTM son realmente necesarios y lo hacen con el uso en su arquitectura de una única puerta que controle tanto los valores que han de ser olvidados como la decisión de si se ha de actualizar el *hidden-state* o no. A diferencia de las LSTM estas cuentan con un único estado de la celda.

Fueron introducidas en 2014 por Kyunghyun Cho y otros investigadores [17].

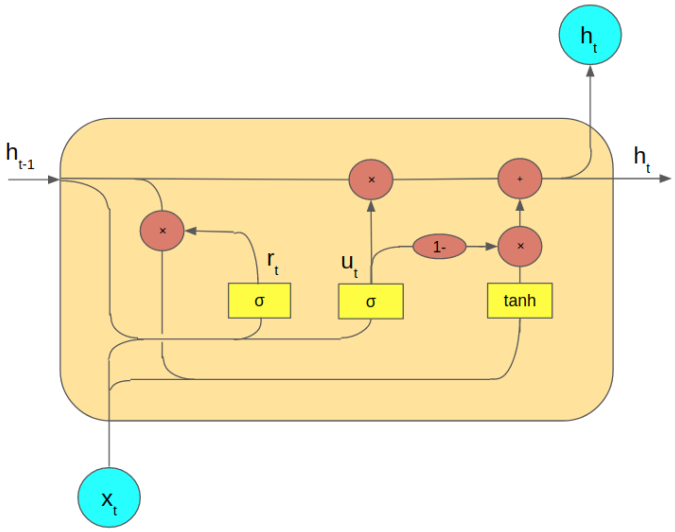

Figura 2.9: Representación de una celda GRU.

La GRU cuenta con dos puertas. La **reset-gate**(r_t) se encarga de seleccionar que partes del **estado** serán usadas para calcular el siguiente **estado**.

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{b}^r + \boldsymbol{U}^r\boldsymbol{x}^{(t)} + \boldsymbol{W}^r\boldsymbol{h}^{(t-1)}\right) \tag{2.29}$$

La **update-gate**(u_t) se encarga de actualizar, o no actualizar (dependiendo del resultado que de la sigmoide) el nuevo **estado**.

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{b}^u + \boldsymbol{U}^u\boldsymbol{x}^{(t)} + \boldsymbol{W}^u\boldsymbol{h}^{(t-1)}\right) \tag{2.30}$$

Por lo tanto el nuevo **estado** será:

$$\boldsymbol{h}^{(t)} = \boldsymbol{u}^{(t)}\boldsymbol{h}^{(t-1)} + (1 - \boldsymbol{u}^{(t)})tanh\left(\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{r}^{(t)}\boldsymbol{h}^{(t-1)}\right) \tag{2.31}$$

Las puertas representadas como superíndice de b, U o W en la notación anterior pueden tomar los siguientes valores:

- r: Es la *reset-gate*.
- u: Es la *update-gate*.

Si no se especifica un valor para q nos estaremos refiriendo a los pesos y *bias* usados para el cálculo de $\boldsymbol{h}^{(t)}$

2.4. Librerías para aprendizaje automático

En esta sección detallamos las librerías más importantes hoy por hoy a la hora de hacer aprendizaje automático. Cabe destacar que todas las librerías que ahora se van a mencionar tienen una implementación para *python*. Este es un lenguaje de programación que hoy en día está tomando mucho protagonismo en el campo del aprendizaje automático.

2.4.1. Scikit-learn.

Scikit-learn [18] es una librería software libre creada por David Cournepeau en Junio de 2007. En esta se implementan funcionalidades útiles para el aprendizaje automático tales como métodos de preprocesamiento de datos, de particionado de los mismos o incluso distintos clasificadores como vecinos próximos, random forest o máquinas de vectores de soporte. Nos da también una implementación del perceptrón multicapa.

2.4.2. Tensorflow.

Tensorflow es una librería para aprendizaje automático implementada en un principio para *python* y ahora también disponible para *javascript*. Fue creada por Google y en 2015 se cambió su licencia a código abierto[19].

Esta librería implementa múltiples funcionalidades de aprendizaje automático y es muy reconocida por su utilidad en la implementación de modelos de redes neuronales. Su funcionamiento se basa en operaciones con *tensores*. Los *tensores*, según Google explica en su documentación[20], son generalizaciones de vectores y matrices a dimensiones más altas.

Su uso se ha popularizado gracias a la versatilidad y facilidad que da a la hora de desarrollar modelos de redes neuronales que se ejecuten sobre GPUs. Esto es muy importante sobre todo a la hora de crear modelos complejos y de entrenar con grandes cantidades de datos ya que reduce en gran medida los tiempos de entrenamiento. Otra de las ventajas a destacar es que el usuario solo tiene que definir el grafo de la red, Tensorflow realizará el algoritmo de retropropagación de manera automática.

2.4.3. Keras.

Keras es una librería de código abierto de redes neuronales, que se implementa sobre otros *frameworks* de aprendizaje automático como Tensorflow, *CNTK* [21] o *theano* [22]. Apareció en 2015 y fue creada por François Chollet.[23]

Tal y como se menciona en su documentación[24], Keras se guía por cuatro principios. Ser amigable para el usuario, modular, fácilmente extensible e implementada en *python*.

2.5. Aplicaciones de las redes recurrentes.

Existen numerosos trabajos en el área de las redes neuronales recurrentes. Entre estas destacan las labores de generación de secuencias, como texto o música, etiquetado de imágenes, traducción, aplicación a *chatbots*, etc.

Una de las empresas que más trabajo está realizando en este campo es Google, sobre todo en el área de **traducción**. Google ha puesto gran esfuerzo en aplicar estos modelos a su traductor, mejorando considerablemente los resultados, evitando la implementación de reglas abstractas que pudieran resolver este problema sin tan buenos resultados. Uno de los puntos más destacados, es que el traductor solo sabe traducir directamente de cualquier idioma al inglés y viceversa, pero consigue dar traducciones precisas entre idiomas distintos al inglés realizando un paso intermedio de traducción al inglés y traducción del inglés al segundo idioma[25].

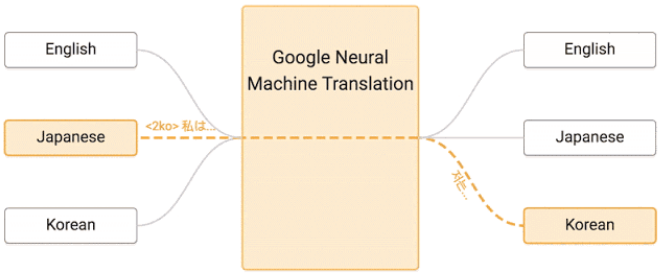

Figura 2.10: Diagrama del funcionamiento del traductor de Google.

La arquitectura del modelo del traductor sigue un patrón *sequence-to-sequence* como el introducido en la figura 2.11, con ocho capas LSTM entre *encoder* y *decoder* para mejorar la precisión[26].

Esta misma empresa también ha desarrollado trabajos en el área de la generación de música. Su proyecto, **Magenta**, es hoy en día el trabajo que mejores resultados ha dado para este problema. Este es un proyecto de investigación y código abierto que explora como el aprendizaje automático puede ser aplicado a la generación de música y arte en general.

En un *post* del apartado de Magenta en la página de *Tensorflow* titulado **Performance RNN: Generating Music with Expressive Timing and Dynamics** se nos muestra como usando un banco de datos de ficheros MIDI y una LSTM se pueden generar melodías polifónicas y expresivas en cuanto a ritmo[27].

También se detalla que para aumentar el número de ejemplos de entrenamiento se incrementa o reduce la velocidad y/o el tono de las melodías. Esta técnica se conoce como *data augmentation* y es muy conveniente en aprendizaje automático para poder disponer de más ejemplos de entrenamiento.

En general, la generación musical se realiza a partir de aprendizaje sobre ficheros de audio en bruto, o sobre ficheros MIDI.

Un *paper* titulado **Music Composition using Recurrent Neural Networks** de los Departamentos de Ingeniería Electrónica y Ciencias de la Computación de la universidad de Stanford muestra como generar música a partir de ficheros de descripción de audio en notación ABC y utilizando modelos *sequence-to-sequence*. Además, su modelo se entrena con un total de 35.000 ficheros de audio [28].

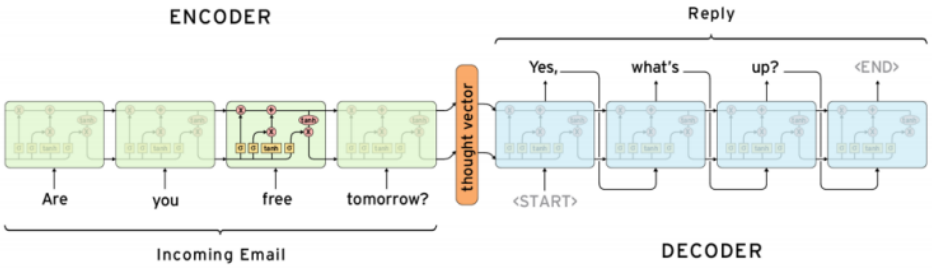

Figura 2.11: Modelo sequence-to-sequence conocido también como encoder-decoder.

Este mismo *paper* también explica que para el caso de los ficheros de audio en bruto es común hacer un mapeo de los datos de entrada a los Coeficientes Cepstrales en las Frecuencias de Mel (MFCCs por sus siglas en inglés) a modo de preprocesamiento de los datos. Estos son

coeficientes para representar los sonidos de un modo más parecido a como el oído humano los percibe [29]. De este modo se discretiza la señal.

Otro área de desarrollo de las redes neuronales recurrentes es el etiquetado de imágenes. En esta destaca un trabajo llamado llamado **Deep Visual-Semantic Alignments for Generating Image Descriptions** [4] que muestra como este tipo de redes pueden utilizarse para el etiquetado de imágenes.

La red combina redes neuronales convolucionales con redes neuronales recurrentes para obtener estos resultados. En general consigue describir lo que en la fotografía se muestra con precisión.

3

Diseño.

En este capítulo se enumeran los experimentos a desarrollar. Comenzaremos explicando los experimentos básicos planteados y tras ello definiremos los experimentos que se llevarán a cabo con el objetivo de la generación de música.

3.1. Generación de texto.

Las primeras pruebas a realizar se basarán principalmente en la generación de texto. Para esto, definiremos modelos inspirados en una de las tres redes recurrentes explicadas en la sección 2.3 y seleccionaremos un banco de datos acorde con nuestras necesidades.

3.1.1. Planteamiento del experimento.

La orientación que se le puede dar a este problema es variada. Puede ser tanto una generación carácter por carácter como una predicción palabra por palabra. Hay otras orientaciones para este problema como son las que llevan a cabo las arquitecturas *encoder-decoder* usadas principalmente para *chatbots*.

El modelo más sencillo es la predicción carácter por caráceter. Pese a que, la orientación palabra por palabra puede dar resultados más coherentes, este planteamiento es más complicado debido a la alta dimensión de los datos de entrada. Hoy en día es posible realizar modelos orientados a la predicción por palabra gracias a los *embeddings*, entre ellos *word2vec* [30]. Estos reducen en gran medida la dimensión de los datos de entrada.

Para este experimento nos quedaremos con la predicción carácter a carácter, dado que el **objetivo** que perseguimos en esta prueba es la de discernir que modelo de red recurrente da mejores resultados en la predicción sobre series temporales. Ver figura 3.1.

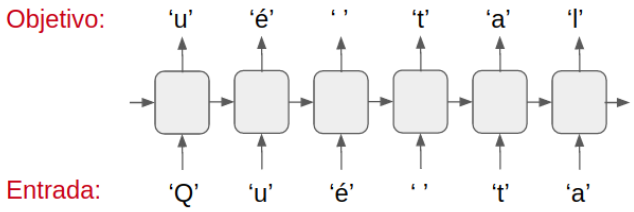

Figura 3.1: Funcionamiento de la predicción por carácter.

Dado que nuestro objetivo es realizar una comparativa entre modelos, definiremos tres modelos distintos. Una RNN básica, una GRU y una LSTM. Entrenaremos estas redes el mismo número de pasos de tiempo y evaluaremos las diferencias en entrenamiento de estos modelos.

La **herramienta** a usar será **tensorflow**. Esta librería introducida en la sección 2.4.2 da una gran versatilidad a la hora de definir modelos de redes neuronales y nos ofrece una implementación de los tres modelos que queremos comparar. Además, comprendiendo el funcionamiento de **tensorflow** nos será más sencillo entender como operan otras librerías como Keras.

Debido a que en **aprendizaje supervisado**, los valores objetivos de los patrones representan categorías, estas categorías son valores discretos que han de ser llevados a un campo numérico que entienda nuestra red. Una representación muy común en redes neuronales es el ***One-hot encoding***[31]. Se basa en relacionar cada categoría con un valor numérico y convertirlo en un vector $[x_1, ..., x_i, ..., x_n]$ en donde n indica el número distinto de clases, en este caso de caracteres, e i indica el valor numérico establecido para la clase a codificar. En este vector todos los valores estarían a 0 a excepción de la posición i que contendría un 1.

Dado que la dimensión de los datos para este problema es pequeña, no se requiere ninguna operación específica sobre los mismos más allá de pasarlos a **One-hot encoding**.

Una vez nuestro modelo esté entrenado, seleccionaríamos una semilla y generaremos más datos a partir de esta. En este caso, la semilla estará formada por un único símbolo.

Una decisión de diseño a la hora de generar fue la de seleccionar como predicción un elemento elegido de manear aleatoria en base a las probabilidades del vector calculado por la red. Esto se realizó para evitar que la red se quedase atascada generando una misma secuencia una y otra vez.

De este experimento se concluyó que pese a que Tensorflow nos da gran versatilidad, su sintaxis es complicada sobre todo si se quieren construir modelos más complejos.

3.2. Experimentos dirigidos a la generación de música.

Para la generación de música habrá dos aproximaciones. La primera, se basará en crear música entrenando un modelo con ficheros de audio en bruto (*.wav*). La segunda aproximación se basará en generar música entrenando el modelo con ficheros de descripción de audio como los *MIDI*.

En este apartado el modelo que se utilizará será el que mejores resultados de aprendizaje haya obtenido en los experimentos explicados en la sección 3.1. La herramienta a utilizar para su implementación será Keras. La decisión de usar esta herramienta viene por lo expresado en el apartado anterior de la complejidad que supone la creación de modelos en Tensorflow. Debido a que esta herramienta es únicamente una capa que se coloca sobre Tensorflow para simplifi-

car la creación de modelos, no deberíamos notar diferencias en el proceso de entrenamiento o generación.

3.2.1. Ficheros de audio en bruto.

Para este experimento se entrenará la red sobre ficheros *.wav*. Los ficheros WAV (Waveform Audio File Format) son un estándar a la hora de almacenar flujos de audio en binario. Normalmente los ficheros WAV almacenan valores muestreados a 44.100 Hz con 16bits por muestra.

La librería ***Scipy*** de código abierto, provee de herramientas para ciencia, ingeniería y matemáticas. En este experimento será de gran ayuda con este problema por su paquete ***wavfile***, que dota de herramientas para la lectura y escritura de ficheros WAV a vectores *numpy* o desde vectores *numpy*.

Debido a que los datos de entrenamiento son una onda de audio, es decir, valores continuos, tendremos dos aproximaciones la primera sería **discretizar** estos datos para reducir su dimensión alta y hacer el entrenamiento posible planteando el problema como uno de regresión logistica, o la segunda que sería no realizar ninguna operación sobre los datos y plantearlo como un problema de regresión.

El planteamiento de entrenamiento y generación es similar al de la sección 3.1. Por ello, entrenamos con un modelo similar al mostrado en la figura 3.1, a partir de una secuencia de valores, predecimos el siguiente. Una vez nuestro modelo esté entrenado, seleccionaríamos una semilla que en este caso será una secuencia y generaremos más datos a partir de esta.

En el caso de que el problema se plantee como una regresión logística elegiremos como predicción un valor elegido aleatoriamente entre el vector de probabilidades al igual que se explicó en el experimento anterior.

3.2.2. Ficheros MIDI.

Los ficheros MIDI (Musical Instrument Digital Interface) son ficheros de audio que, a diferencia de los ficheros WAV no contienen sonido digitalizado. Sin embargo, contienen una descripción tanto de las notas o acordes de la melodía como de la duración de los mismos. Además pueden contener más de una pista con distintos instrumentos.

Para simplificar, en nuestro experimento, en el caso de que la melodía tenga más de un instrumento se seleccionará únicamente uno. Se extraerá a continuación de la melodía unicamente acordes, notas y duraciones de los mismos. Obtenemos los vectores *One-hot* y entrenamos.

Usaremos la librería ***music21*** [32] desarrollada en el MIT que provee de la funcionalidad necesaria para extraer las distintas características de este tipo de archivos.

Tanto en entrenamiento como en generación deberemos asociar de algún modo tanto las notas y acordes con sus duraciones de tal manera que la red pueda aprender también esta dependencia.

Una vez la red esté entrenada seguiremos el mismo esquema que hemos venido aplicando para el resto de experimentos para generación de nuevos datos.

4

Desarrollo.

4.1. Desarrollo de las pruebas

4.1.1. Generación de texto.

Para esta prueba, tal y como mencionamos en el capítulo 3 utilizaremos Tensorflow. Implementamos tres modelos de red: Una Vainilla RNN, una GRU y una LSTM. Nuestro objetivo será realizar generación de texto carácter a carácter y comprobar el comportamiento de cada una de las redes.

Tensorflow nos ofrece clases como *BasicRNNCell*, *GRUCell* y *LSTMCell* para implementar modelos de estos tipos. Tras cada uno de estos modelos aplicamos una capa *softmax* (ver 2.22) final de tal modo que obtengamos como valores de salida una lista de probabilidades por clase.

Como datos de entrenamiento hemos elegidos el libro entero de *El Quijote* en formato *.txt*[1]. Este cuenta con un total de 2.073.255 caracteres de los cuales 80 son únicos. Además se han seleccionado los siguientes hiperparámetros.

Constante de aprendizaje	0,01
Número de neuronas en la capa oculta	100
Longitud de la secuencia de entrenamiento	50
Número de pasos de entrenamiento	500.000

Tabla 4.I: Hiperparámetros del modelo. Dado que en este experimento únicamente queremos ver las diferencias de las distintas redes, se han elegido únicamente estos hiperparámetros para todos los modelos.

El optimizador elegido será el ***adam***. Este es un método eficiente de optimización estocástica que únicamente requieren gradientes de primer orden [33]. Este método deriva del algoritmo de *descenso de gradiente estocástico* que se desarrollo por la necesidad de acelerar el entrenamiento de las redes neuronales reduciendo el tiempo de cálculo del algoritmo tradicional de descenso de gradiente 2.2.4.

[1] https://www.gutenberg.org/cache/epub/2000/pg2000.txt

Como se explicó en la sección 2.3.2 la RNN Básica tiene problemas de *vanishing* y *exploding gradients*. Por tanto se comparó el entrenamiento de esta red con y sin aplicar ***gradient clipping***.

Para los otros dos modelos de red, llevamos a cabo el mismo entrenamiento y comparamos los resultados con los de la red anterior. Los resultados se pueden ver en la sección 5.1.

Aparte de observar el proceso de entrenamiento de estos tres modelos, se probó a generar texto con los modelos entrenados con El Quijote y se llevó a cabo generación de código C tras haber entrenado la red con código fuente del *kernel* de Linux.

4.1.2. Generación de música a partir de audios en bruto.

Tal y como introducimos en la sección 3.2.1, el siguiente experimento perseguía la generación de música a partir de ficheros de audio en bruto.

Para esto, en primer lugar necesitábamos obtener un conjunto de datos. Recordemos que el muestreo de los ficheros *wav* es de 44.100 Hz, por tanto el fichero guarda 44.100 valores por segundo. Esto significa que una melodía de 2 minutos contendría un total de $2min \cdot 60\frac{seg}{min} \cdot 44.100\frac{valores}{seg} = 5.292.000$ valores en total. Son datos más que suficientes para probar nuestro modelo.

Como fichero de entrenamiento hemos utilizado una melodía de 1 minuto y 44 segundos de duración [2]. Para leer el archivo utilizaremos el módulo ***scipy.io.wavfile***. Este nos dota de la función ***read*** que a partir del nombre del fichero WAV nos devolverá la tasa de muestreo y un vector unidimensional o bidimensional, conteniendo los distintos valores de la onda, dependiendo de si el fichero de audio cuenta con una o dos pistas. Nos quedaremos siempre solo con una pista.

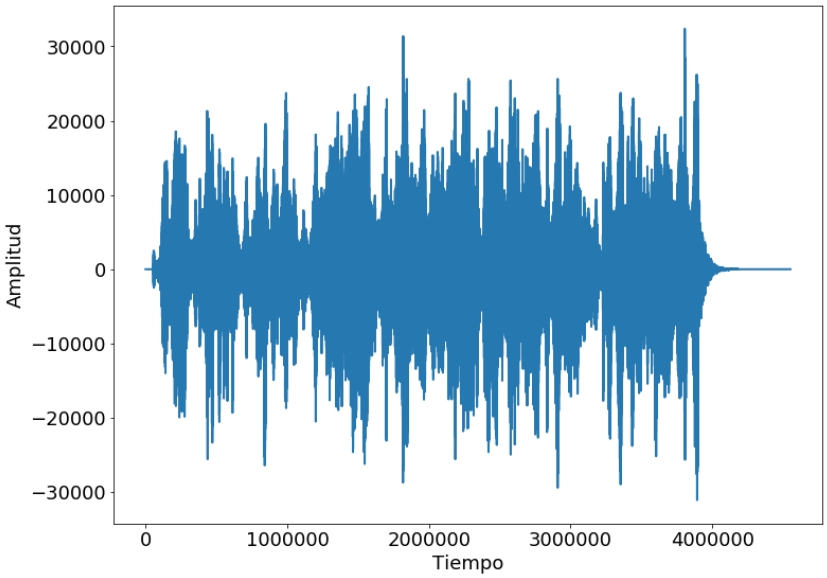

Figura 4.1: Onda del fichero WAV usado como datos de entrenamiento.

Podemos observar que este fichero (Ver figura 4.1) cuenta con 4.551.841 valores de los cuales **42.156 son únicos**. Como vemos, este conjunto de datos necesita de algún tipo de preprocesamiento ya que manejar 42.156 clases en un vector *One-hot* sería impracticable.

[2] https://soundcloud.com/user-494001124/classique

Número de neuronas en la capa oculta	100
Longitud de la secuencia de entrenamiento	100
Número de pasos épocas de entrenamiento	30

Tabla 4.II: Hiperparámetros del modelo. Estos hiperparámetros se han elegido para realizar la prueba de tal modo que el equipo utilizado pudiese manejarlos. Se probó con más cantidad de neuronas en la capa oculta o secuencias mayores sin éxito debido a que los tiempos de entrenamiento en el equipo usado se volvían inasumibles.

El siguiente paso sobre los datos, es discretizarlos reduciendo su dimensión. En primer lugar eliminamos los silencios del principio y final de la melodía. Tras ello, dividimos todos los valores entre un determinado número y nos quedamos únicamente con la parte entera del resultado. De este modo reduciremos el número de posibles valores que la melodía puede tomar.

Este valor por el que dividir habrá de ser seleccionado con cautela de tal modo que el valor sea lo suficientemente alto como para reducir la dimensión del vector de clases de la onda y lo suficientemente bajo para no reducir la calidad del audio. En nuestro caso el valor elegido ha sido 300 ya que este valor es grande pero no se aprecia una pérdida de calidad al oído.

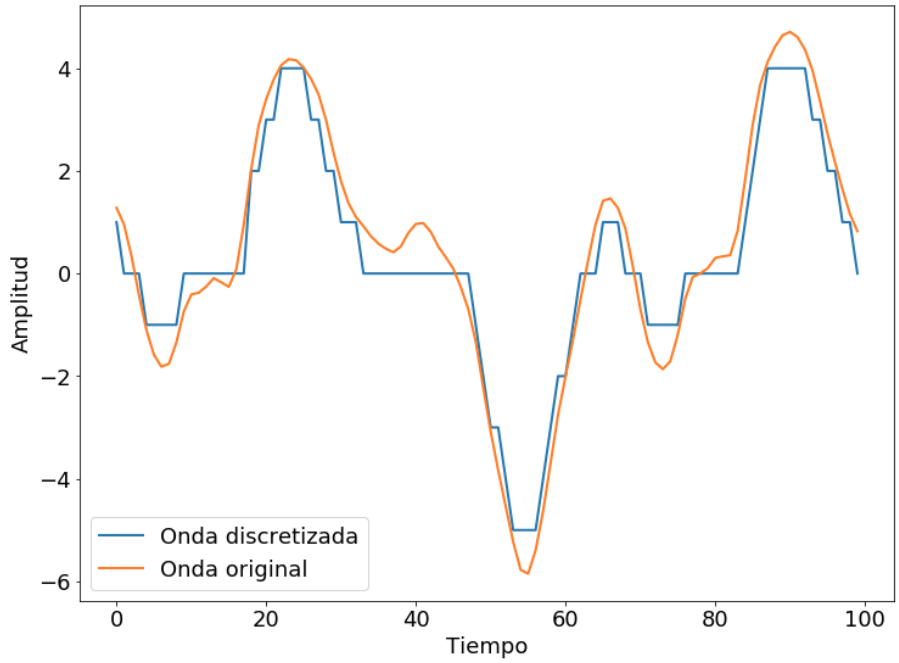

Figura 4.2: Fragmento de la onda sin discretizar y tras discretizarla. Se recuerda que los valores de la onda original son 300 veces mayores que los de la onda discretizada, estos se han reescalado para poder realizar la comparativa mostrada en esta figura.

Tras eliminar los silencios y discretizar la onda el vector de datos de la melodía queda con una longitud total de 3.983.918 valores de los cuales **208 son únicos**. Un ejemplo de esto se puede ver en la figura 4.2.

Construimos una red LSTM usando Keras con una capa *softmax* final. Los hiperparámetros seleccionados son los siguientes:

Al igual que en el modelo de texto, las secuencias de entrada a la red se desfasan en una unidad de tiempo de las secuencias objetivo, de tal modo que los valores objetivos para cada valor de entrada sean el valor siguiente de la secuencia. Esto se muestra en la figura 4.3.

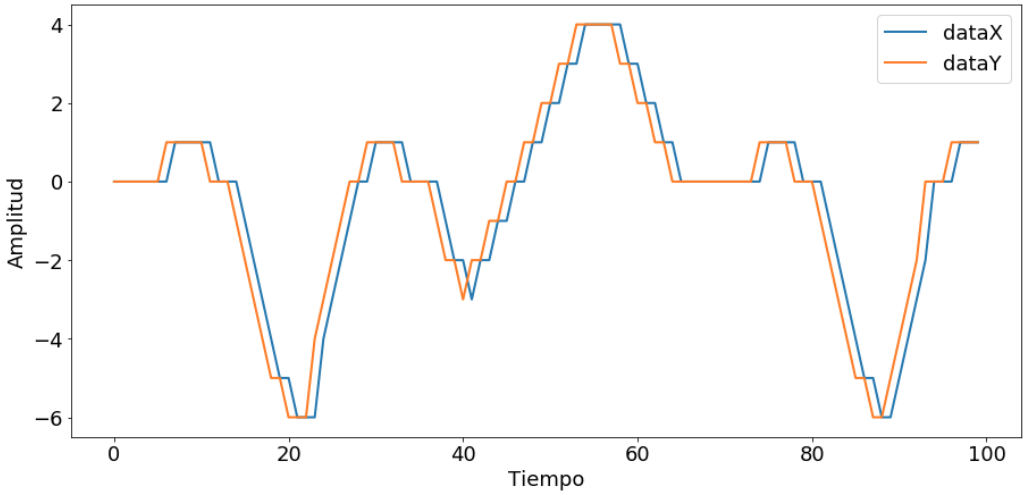

Figura 4.3: Desfase que sufre la secuencia de entrada con respecto a la secuencia objetivo. Se prepara de tal modo que cada valor de entrada dataX tiene como valor objetivo el siguiente valor de la secuencia. Esta secuencia de valores objetivos se encuentra en dataY.

Tras entrenar el modelo para generar audio seguimos el mismo esquema que el mencionado en la sección anterior. Seleccionamos una secuencia como semilla y a partir de ese predecimos el siguiente, el valor generado se concatena en una lista con la semilla, y volvemos a generar otro valor. Este proceso se realiza hasta llegar a tamaño de secuencia determinado.

Debido a que los valores leídos para entrenamiento fueron discretizados dividiéndolos por 300, los valores generados se encontrarán también en ese rango. Para escribirlos de nuevo en un fichero WAV debemos devolverlos a su rango original multiplicando toda la secuencia generada por 300.

Utilizaremos la función ***write*** del módulo ***scipy.io.wavfile*** para escribir esta lista de datos a un fichero WAV. Como entrada a esta función tendremos que especificar tanto los datos como la tasa de muestreo y el nombre del fichero a generar.

En el caso de plantear el problema como un problema de regresión no sería necesario llevar a cabo todo el proceso de discretización anteriormente mencionado.

Otro de los modos de llevar a cabo esta prueba será la de dividir la onda en bloques de un cierto número de valores y hallar la **transformada de Fourier** de cada bloque sustituyendo así el método de discretización por este. La transformada de Fourier pasa una onda del dominio del tiempo al dominio de la frecuencia. El objetivo será predecir a partir de un bloque de frecuencias el bloque siguiente.

Para este problema utilizaremos una LSTM pero lo plantearemos como un problema de **regresión** sin discretizar los valores de los bloques y obtener sus vectores *One-hot*. En este caso intentaremos minimizar el error cuadrático médio.

Una vez generamos todos los bloques realizamos la inversa de la transformada de Fourier en cada uno de estos para volver al dominio del tiempo y poder guardar el WAV.

4.1.3. Generación de música a partir de ficheros MIDI.

Como definimos en la sección 3.2.2 en este experimento trataremos de generar música a partir de ficheros MIDI.

Los datos seleccionados han sido un conjunto de ficheros MIDI que se han fusionado para formar uno solo y con el que entrenar. El método ***parse*** del módulo ***music21.converter*** extrae los datos de un fichero de este estilo.

```
from music21 import converter

midi = converter.parse("data/clasica/1812-rev.mid")
for element in midi.recurse():
    print(element)
```

```
<music21.stream.Score 0x7fd9a202fcc0>
<music21.stream.Part 0x7fd9ae015128>
Trumpet
Trumpet
<music21.tempo.MetronomeMark adagio Quarter=56.0>
E- major
<music21.meter.TimeSignature 3/4>
<music21.stream.Voice 0x7fd9ae030320>
<music21.note.Rest rest>
<music21.note.Note E->
<music21.chord.Chord F5 B-4 D4 D5>
<music21.chord.Chord G5 B-4 E-4 E-5>
<music21.chord.Chord E-4 B-4 G4 E-5>
```

Figura 4.4: Elementos extraidos de un fichero MIDI por la librería de *music21*.

De estos objetos nos quedaremos unicamente con notas y acordes y extraeremos la duración de cada uno de ellos. A continuación debemos establecer una representación alternativa de estos objetos que sea más fácilmente almacenable en forma de secuencia.

- Notas: Para todas las notas guardaremos únicamente el nombre de la misma en una cadena de caracteres.
- Acordes: Debido a que los acordes están formados por numerosas notas, crearemos una cadena de caracteres que guardará el **orden normal** del acorde siendo esta una manera de representación de los acordes según la teoría musical [34]. Dado que esto es una lista de números los guardaremos en una cadena de caracteres separados por un punto.

$$< music21.chord.Chord\ E4\ G4 >\equiv \text{“4 . 7”}$$

- Duraciones: Las duraciones serán extraídas tanto de notas como de acordes. Para estos guardaremos su nombre que es del estilo *zero, quarter, eighth, 16th,* etc. En los ficheros MIDI también podemos encontrar tipos de duración más abstractos, como *inexpressible* o *complex*. En el caso de que nos encontremos con alguno de estos asumiremos que la duración es de *quarter*.

El modo de asociar la nota o acorde con su duración será colocarlos en serie, es decir, en la secuencia de entrenamiento a generar, por cada nota u acorde de la misma se colocará a continuación su duración.

```
['D5', '2048th', 'C#5', '2048th', 'E5', '2048th', 'C#5', '2048th', 'A4', '2048th', 'D5', '2048th', 'E5', '2048th',
 'D5', '2048th', 'A4', '2048th', 'E5', '2048th', 'F5', '2048th', 'E5', '2048th', 'A4', '2048th', 'D5', '2048th', 'E
5', '2048th', 'D5', '2048th', 'A4', '2048th', 'C#5', '2048th', 'B4', '2048th', 'A4', 'eighth', 'G4', '2048th', 'A
4', '2048th', 'B4', '2048th', 'C5', '16th', 'C#5', 'eighth', 'D5', '2048th', 'E5', '2048th', 'F5', '2048th', 'G5',
 '2048th', 'F5', '2048th', 'E5', '2048th', 'G5', '2048th', 'F5', 'eighth', 'A5', 'quarter', 'D5', '2048th', 'C#5',
 '2048th', 'D5', '2048th', 'G5', '16th', 'F5', 'eighth', 'E-5', '2048th', 'D5', '2048th', 'F5', '2048th', 'E-5', 'e
ighth', 'D5', 'eighth', 'C#5', 'eighth', '2.3', 'quarter', '2.3', 'quarter', '2.3', 'quarter', 'D5', '2048th', 'E
5', 'eighth']
```

Figura 4.5: Secuencia construida con la sintaxis definida con anterioridad. Este formato de secuencia es el que la red recibirá como entrada de entrenamiento

Número de neuronas en la capa oculta	100 y 256
Longitud de la secuencia de entrenamiento	50
Número de pasos épocas de entrenamiento	300

Tabla 4.III: Hiperparámetros del modelo. Se harás dos pruebas distinto número de neuronas en la capa oculta, la longitud de la secuencia se dejará fija a ese valor para que no se dispare mucho el tiempo de entrenamiento,

Al igual que en los experimentos anteriores, tendremos que crear un vector *One-hot* para cada elemento de la secuencia y preparar las secuencias objetivo desfasándolas en un valor con respecto de las de entrada.

Tras esto entrenamos y probamos a generar resultados. Debido a que la codificación que se le ha dado a las secuencias de entrenamiento es nota/acorde seguido de la duración de dicha nota o acorde puede darse el caso de que a la hora de generar nuevas secuencias tras una nota/acorde no se prediga su duración y se prediga otra note/acorde. En el caso de que en la secuencia aparezca esta situación, a la hora de crear el fichero MIDI estableceremos para la nota/acorde que carece de duración una estándar de *un cuarto (quarter)*. Si por contra se generan dos duraciones seguidas, la segunda de ellas se ignorará.

En general, salvo en casos en los que el modelo tiene poco entrenamiento esta corrección no es necesaria ya que la red aprende correctamente esta secuencia de notas/acordes seguidas de duraciones. Esto se muestra en la figura 5.8.

Cabe destacar que otra de las codificaciones para crear la secuencia de entrenamiento que se probó *sin buenos resultados* fue la de construir vectores *One-hot* distintos para el conjunto de las notas/acordes y el conjunto de las duraciones y fusionar ambos siendo estos vectores los que la red trataría de aprender para cada elemento de la secuencia. A modo de ejemplo, en el caso de que la nota sea $C4$ y su duración sea *quarter* hallamos primero los vectores *One-hot* correspondientes, supongamos que para $C4$ este es $[0, 0, \mathbf{1}, 0, 0, 0]$ y para *quarter* es $[\mathbf{1}, 0, 0]$. En ese caso el vector que trataría de aprender la red sería la fusión de ambos, es decir, $[0, 0, \mathbf{1}, 0, 0, 0, \mathbf{1}, 0, 0]$.

5

Resultados de los experimentos.

En este capítulo se hará un análisis de los resultados obtenidos a partir de las arquitecturas de red mostradas en la sección anterior. En primer lugar detallaremos algunas pruebas básicas llevadas a cabo con textos para posteriormente pasar a describir los resultados de las pruebas llevadas a cabo con audios.

5.1. Pruebas básicas

Las primeros experimentos realizados han sido llevados a cabo para poder discernir que modelo de red de los tres explicados con anterioridad era mejor para un problema de este estilo. Los modelos evaluados han sido la *Vainilla RNN*, la *GRU* y la *LSTM*. Se han realizado sobre un fichero de texto.

5.1.1. Primeras pruebas.

Para intentar decidir que modelo de red es mejor compararemos dos distintas medidas. La primera de ellas es la evolución del *coste* a lo largo del proceso de entrenamiento y la segunda el tiempo que la red tarda en entrenar un número de terminado de *pasos*. El número de **pasos de entrenamiento** máximos que hemos establecido ha sido de 500.000. Los **pasos de entrenamiento** son el número de veces que se alimenta a la red con un par (entrada, objetivo). Dado que el problema a resolver es un problema de generación, se ha considerado que no tiene sentido establecer un conjunto de datos de *test*. Estableceremos otras métricas para comprender como de buenos son los resultados.

La primera prueba realizada se llevó a cabo con una *Vainilla RNN*. La clase que *TensorFlow* nos facilita para la utilización de este tipo de red es la *BasicRNNCell*.

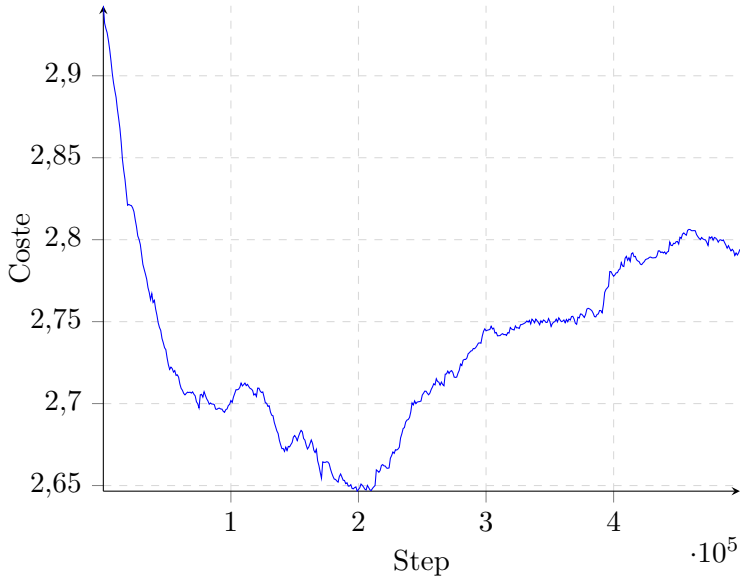

Figura 5.1: Evolución del coste durante el entrenamiento con una Vainilla RNN.

Podemos observar que el aprendizaje no es lo que esperaríamos de una red así, en la gráfica se puede observar que, a parte de que el descenso del coste se ve irregular, a partir del *step* 200.000 el coste vuelve a crecer, y no da señales de volver a bajar. Es decir, la red no está aprendiendo.

Como ya desarrollamos en la sección 2.3.2 la *Vainilla RNN* suele tener problemas de *vanishing* o *exploding gradients*. Uno de las consecuencias de esto es lo que podemos ver en el gráfico anterior, la red no logra aprender.

Aplicaremos *gradient clipping* para observar si la red consigue evitar estos problemas.

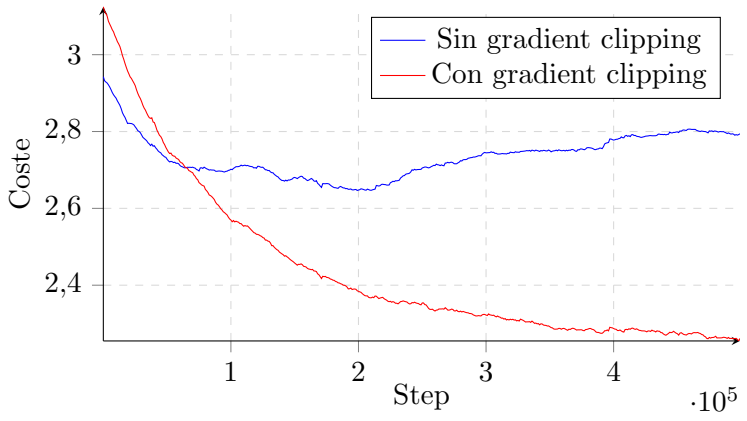

Figura 5.2: Diferencias en las evoluciones de la Vainilla RNN sin y con *gradient clipping*.

En la gráfica anterior podemos observar que realmente el *gradient clipping* funciona y evita que el gradiente explote o desaparezca. Se puede observar que la red si logra aprender cuando aplicamos esta solución. El coste se decrementa a cada paso y parece mostrar que tras la iteración 500.000 vaya a seguir la misma tendencia.

Sin clipping	Con clipping
0:59:36 h	1:00:17 h

Tabla 5.I: Diferencia de tiempos de entrenamiento realizando 500.000 pasos entre las dos variantes de la Vainilla RNN (en horas). Llevado a cabo en una máquina con 8GB de RAM y un procesador Intel Core i5 de cuarta generación a 1,7 GHz.

Como se puede ver en la tabla 5.I prácticamente no hay diferencia en el tiempo de ejecución del entrenamiento entre ambas redes.

Comparamos ahora la *GRU* y la *LSTM* con la *Vainilla RNN* con *gradient clipping* ya que es este la variante de las dos que como hemos visto obtenía mejores resultados.

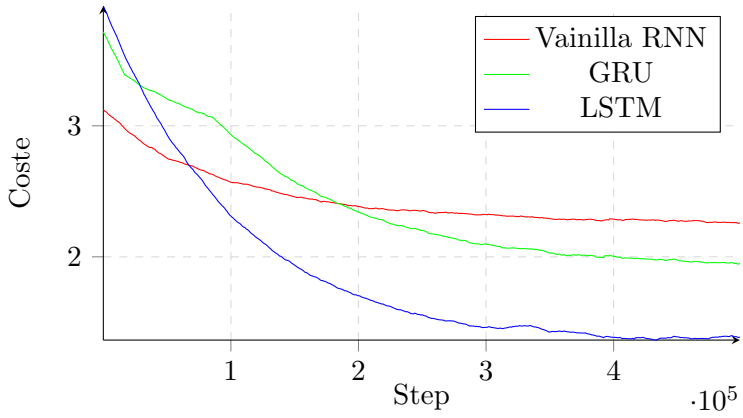

Figura 5.3: Diferencias en las evoluciones de la *Vainilla RNN*, la *GRU* y la *LSTM*

Como ya desarrollamos en la sección 2.3 la LSTM y la GRU evitan los problemas de *exploding* y *vanishing gradients* y consiguen aprender mejor dependencias temporales largas gracias a la estructura de puertas de sus celdas.

Esto se puede ver plasmado en el gráfico de la figura 5.3, ambas tanto *LSTM* como *GRU* consiguen llegar a valores del coste más bajos que la *Vainilla* y su gráfico de aprendizaje es también más suave.

Vainilal RNN	GRU	LSTM
1:00:17 h	1:56:58 h	2:14:28 h

Tabla 5.II: Diferencia de tiempos de entrenamiento realizando 500.000 pasos entre la Vainilla RNN, la GRU y la LSTM (en horas)

Aún con los buenos resultados de aprendizaje, su estructura interna las vuelve más lentas debido a que el número de cálculos se incrementa. Esto se puede ver plasmado en la tabla de arriba.

Entre la *GRU* y la *LSTM* podemos extraer que la *LSTM* es mejor a la hora de aprender, pero que esta ventaja con respecto a la *GRU* la hace ser más lenta.

De aquí podemos concluir que dependiendo de lo que más nos interese a la hora de resolver nuestro problema podremos elegir una u otra. Si necesitamos mayor velocidad podemos ceder en precisión y escoger una *GRU* o en el caso contrario elegir la *LSTM* y perder velocidad.

Dado que la *LSTM* parece ser la que mejor aprende, probamos a generar un texto a partir de lo aprendido por esta red.

```
Por dalada naturaleza Sancho Panza y es la
dasen dijo:

-Vengo a esta acabo aconteria
sida entre
para lo malancia
me hare prason respondiendo. Toda
prechua una mano y anda tenia, que por no estretonardas sepa y de lo que continan de uno yo de solo gode y rinciere
respondia; y, a parecer eso que, como no podria o la tener mie de Dios duscreesiajico, y a la insula diegarla menor
a como en vojas dichos seales
caballeres.

¿Que dignad del liderz si ese cual muy aventura con vueltada que, aunque por aquella minla la padridosir, porque no
s entraba capito. Y,,vimiendo a lo que dijese de su sandripana de se le diase, raritas, que
yo podemos echa, honesticios exauteas testidad con don Quijote a los
dejacerle pasarme, para una temerona su heis, y bien no sedrais tan
galo? ¿No ha de suplepla, es molido milicuinto hermina, peersaros a mi
galanzaguro?
```

Figura 5.4: Texto generado por una red *LSTM* entrenada con el libro de El Quijote y orientada a predicción carácter a carácter

La figura 5.4 muestra un fragmento de un texto generado por el modelo mencionado previamente. Este surge a partir de la primera letra que se muestra, en este caso una **P**, esta sería la semilla.

Pese a que muchas de las palabras no tienen significado debido a la orientación a predicción por carácter aplicada, podemos darnos cuenta que todas las palabras son legibles, es decir, no aparecen, en general, fonemas que sean imposibles en nuestro lenguaje.

Se puede ver que la red genera un texto con palabras no muy largas y separadas por espacios, a parte utiliza signos de puntuación a lo largo del texto. Otra virtud del modelo es que este ha aprendido a generar conjuntos de palabras que en el texto de entrenamiento aparecen mucho como es *don Quijote* o *Sancho Panza*.

Además podemos destacar que la red ha podido aprender la estructura para introducir diálogos que se utiliza en literatura, esto se puede ver en el primer y segundo párrafo. Otra de las virtudes a destacar, que puede verse en el tercer párrafo, es que la red ha aprendido la estructura de las preguntas. Podemos ver como tras abrir un signo de interrogación la red acaba cerrándolo.

Los resultados obtenidos a partir del código del *kernel* de *linux* se muestran en la figura B.1 en el anexo B. Los datos usados para entrenar han sido el fichero *fork.c* del mismo [1].

Se puede observar que pese a que este código no va a compilar, la red ha comprendido algunas de las reglas del lenguaje de programación. Por ejemplo, sabe abrir y cerrar corchetes, ha comprendido la estructura de indentación, sabe abrir y cerrar los símbolos que introducen los *includes* y acaba las lineas con punto y coma. La semilla usada ha sido el símbolo #.

5.2. Pruebas con música

Las pruebas mostradas en esta sección han sido dirigidas a la generación de música. Las aproximaciones llevadas a cabo han sido dos.

La primera ha sido tratar de generar audio tras entrenar redes recurrentes con **ficheros de audio en bruto** más específicamente ficheros con formato *.wav*.

La segunda aproximación ha sido tratar de generar audio entrenando la red con ficheros *midi*. Una breve explicación de lo que es un fichero *midi* se muestra en la subsección 3.2.2.

[1] https://github.com/torvalds/linux/blob/master/kernel/fork.c

5.2.1. Ficheros de audio en bruto.

El modelo usado es el mismo que hemos aplicado para el resto de problemas. Utilizamos una red *LSTM* y a partir de una secuencia intentamos predecir el siguiente elemento de la misma. Desarrollamos los resultados obtenidos con el método de discretización de la onda dividiendo por un determinado valor tal y como se desarrolló en la sección 4.1.2.

La red muestra la curva de aprendizaje mostrada en la figura 5.5.

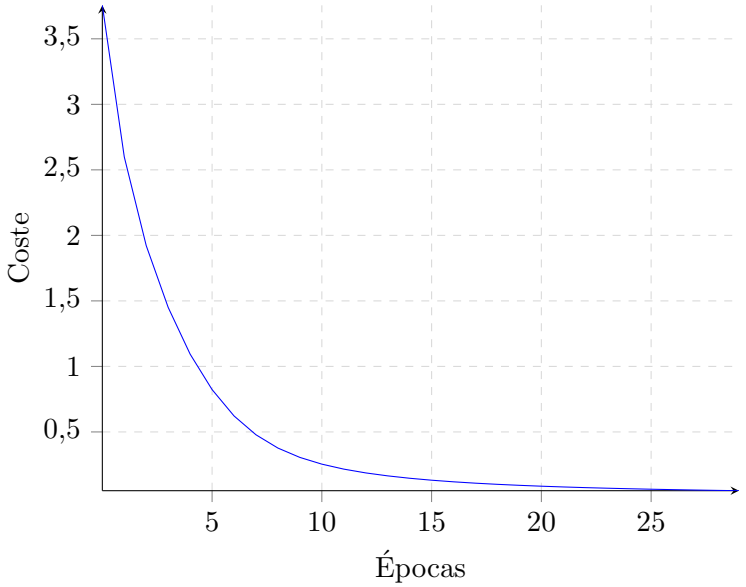

Figura 5.5: Evolución del coste durante las 30 épocas de entrenamiento con el fichero de audio en formato WAV.

Elegimos una secuencia como semilla, siendo esta un secuencia elegida al azar de los datos de entrenamientos, y generamos audio a partir de ella. Se han generado 100.000 valores a partir de esta semilla.

El fichero de audio generado consta de dos segundos de audio en los que únicamente pudimos escuchar un pitido. Una vista más en detalle a este fichero generado nos revela lo siguiente.

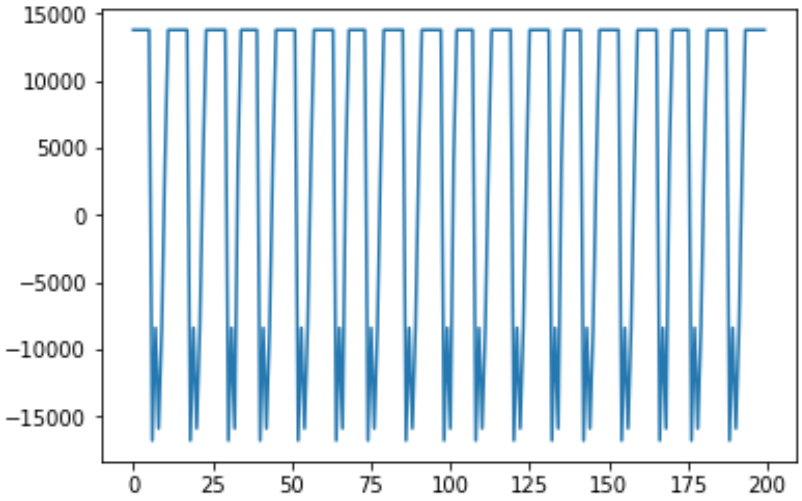

Figura 5.6: Vista más en detalle del fichero.

Se puede percibir que la onda generada se repite en el tiempo. Esto es lo que produce el pitido. Podemos por lo tanto afirmar que la red ha aprendido ya que a partir de una semilla ha logrado generar un patrón que se repite en el tiempo.

Como anteriormente hemos explicado, para el audio generado por la red se han predicho 100.000 valores, y con estos solo hemos podido llegar a unos 2 segundos de audio. Recordemos también que la tasa de muestreo en los WAV es de 44.100 Hz, es decir, 1 segundo de audio está formado por una secuencia de 44.100 valores.

El modelo está entrenado con series de 100 valores lo que equivale a **2,27 milisegundos** de melodía. Una de las hipótesis por la que suponemos que no se están obteniendo los resultados esperados es que dentro de esas secuencias de 2 milisegundos, la red no esté obteniendo mucha información relevante de la melodía. Se intentaron secuencias más largas pero los tiempos de entrenamiento se volvieron impracticables en nuestro equipo.

Además de esto, hemos de recordar, que estamos realizando el entrenamiento sobre ondas de audios que contienen ruido que puede estar interfiriendo en nuestro aprendizaje

Probamos, a continuación a realizar la misma prueba planteando el problema primero como un problema de **regresión** y posteriormente con el método de la **transformada de Fourier** explicado en 3.2.1.

Estas pruebas al igual que el caso anterior producen un resultado periódico que se puede escuchar como un pitido.

Creemos que con un equipo más potente se pudiesen haber obtenido mejores resultados al haber podido probar con secuencias más grandes y modelos con más neuronas en la capa oculta o incluso más capas y sobre todo, con más datos de entrenamiento.

Dado que las capacidades que deberíamos suministrar en la etapa de entrenamiento no son satisfacibles por el equipo que estamos usando probaremos una aproximación con la que podamos obtener mejores resultados.

5.2.2. Ficheros MIDI.

En este caso, hemos entrenado una red como la explicada en el capítulo anterior, con un total de **once** sonatas de Chopin en formato MIDI. La curva de aprendizaje obtenida para este modelo se muestra en la figura 5.7.

Tras probar a generar música, elegimos una semilla formada por una nota/acorde y una duración. Con esta semilla probamos a generar una secuencia. La música generada puede consultarse online[2].

```
['C2', '16th', 'C4', '16th', 'G2', '16th', 'E-3', '16th', 'F3', '16th', 'G4', '16th', 'C#4', '16th', 'F4', '16th',
 'G#4', '16th', 'F#4', '16th', 'F#3', 'eighth', 'F#4', '16th', 'F#4', '16th', 'A4', '16th', 'A3', '16th', 'E4', 'ei
ghth', 'C#4', '16th', 'C3', '16th', 'B-3', 'eighth', 'G#4', '16th', 'A4', '16th', 'A4', 'eighth', 'F#4', '16th', 'F
#3', 'eighth', 'A4', '16th', 'E4', '16th', 'C#5', 'eighth', 'C#4', '16th', 'F#4', '16th', 'E4', '16th', 'E2', '16t
h', 'C#5', '16th', 'G#3', 'eighth', 'E-5', '16th', 'C#4', 'eighth', 'B-4', '16th', 'A4', '16th', 'G#3', 'eighth',
 'D3', '16th', 'C#4', 'eighth', 'F#4', '16th', 'B-3', 'eighth', 'C#5', '16th', 'F4', '16th', 'E3', 'eighth', 'E-5',
'16th', 'B-4', 'eighth', 'E5', '16th', 'F#4', '16th', 'A3', 'eighth']
```

Figura 5.8: Secuencia generada por la red tras 300 épocas de entrenamiento.

Sin ni siquiera escuchar la melodía ya podemos observar algo en claro sobre el aprendizaje y la generación que la red ha realizado mirando la figura 5.8. Se puede ver que el modelo ha

[2]https://soundcloud.com/javier-rom-n-1/sets/music-generated-by-an-lstm-trained-with-11-chopins-sonatas

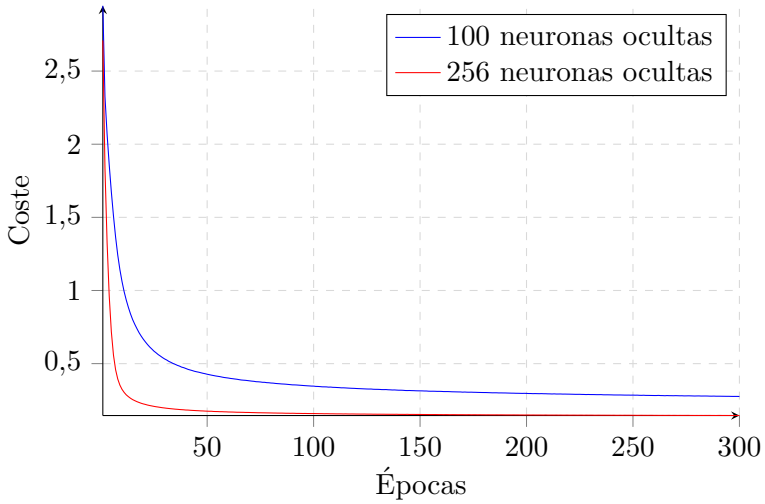

Figura 5.7: Evolución del coste durante las 300 épocas de entrenamiento para las once sonatas de Chopin.

aprendido prácticamente a la perfección que tras una nota/acorde ha de aparecer una duración de la misma.

Un modo en el que podemos medir como de ordenada es una pieza generada es mediante una herramienta que nos ofrece la **teoría de la información** llamada la **entropía** [35] [36]. La entropía en la teoría de la información mide la incertidumbre de un sistema. Esto se puede interpretar como el desorden de un sistema dado que a mayor incertidumbre menos estructura habrá en este.

$$H = -\sum_{i=1}^{D} p_i \cdot log_D(p_i) \tag{5.1}$$

En esta la D representa la **diversidad** siendo esto el número de símbolos distintos que el sistema a evaluar (en nuestro caso las melodía generada) puede contener.

La entropía toma valores entre $[0, 1]$ siendo 0 el máximo grado de certeza y **1 el máximo grado de incertidumbre**.

Para medir como de buenos son los resultados de nuestra red con esta métrica generamos **treinta** melodías para diferentes épocas de entrenamiento, y comparamos la entropía media de cada época con la entropía media del conjunto de datos de entrenamiento. La entropía media del *dataset* de entrenamiento es de 0,221029023754 y la entropía de melodías que se generasen aleatoriamente es 1.

Melodías	**100 Neuronas**	**256 Neuronas**
1 época	0,570651939833	0,598270962568
150 épocas	0,465126038596	0,489132773163
300 épocas	0,454361321447	0,470173795805

Tabla 5.III: Entropía de los distintos conjuntos de datos. Se compara la entropía media del conjunto de datos original con la entropía media de los generados. Se recuerda que la entropía de una melodía aleatoria valdría 1.

Se puede observar en la figura 5.III que a medida que aumentan las épocas de entrenamiento, la entropía de las melodías generadas se van reduciendo. Esto demuestra que las melodías van

siendo menos aleatoria y por tanto que la red está aprendiendo a generar música a partir de lo entrenado. Algo sorprendente es que con una única época la entropía se reduce considerablemente. Otra cosa que cabe destacar es que parece que pese a que el aprendizaje sea más rápido con 256 neuronas en la capa oculta, como se muestra en la figura 5.7, las melodías generadas tienen una entropía ligeramente mayor a aquellas generadas con el modelo de las 100 neuronas en la capa oculta, por lo que tienen un poco menos de estructura. Aún así se puede observar que las variaciones son muy ligeras.

Aúnque sepamos que las melodías generadas tienen cierta estructura. ¿Cómo podemos saber que los fragmentos que mejor suenen de estas no son fruto de un **sobre-aprendizaje** de la red?. La **correlación cruzada** es una métrica que nos permite ver cómo de parecidas son dos señales de audio, o también como de desfasadas están dos señales iguales.

En nuestro experimento nos basaremos en el procedimiento realizado en este *paper* de la Universidad de Tartu[37] sobre los datos obtenidos tras entrenar 300 épocas. Hallaremos la *transformada de Fourier* de las ondas de audio de las melodías de entrenamiento y las generadas, y aplicaremos la correlación cruzada.

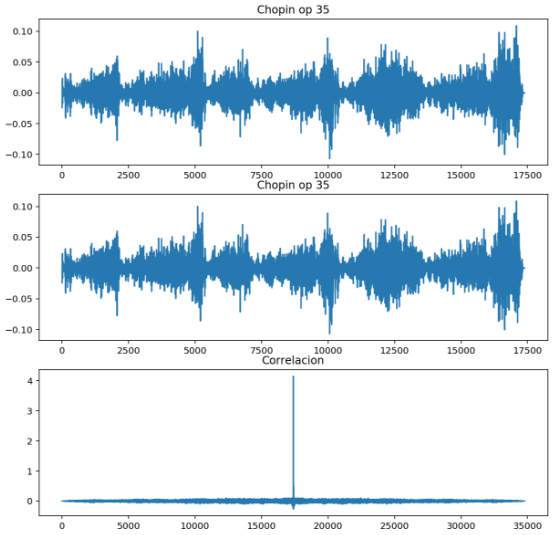

Figura 5.9: Auto-correlación de una onda consigo misma.

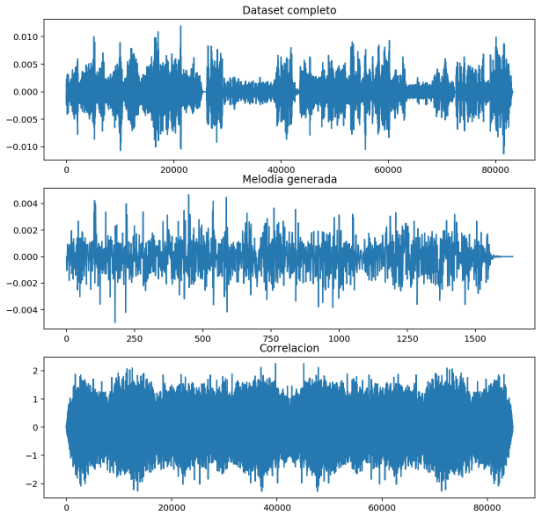

Figura 5.10: Correlación de una onda con otra distinta.

En las dos imágenes anteriores se aprecia que cuando se realiza la correlación cruzada de una onda consigo misma dicha métrica se dispara. En el caso opuesto, la correlación se mantiene en el mismo rango de valores todo el tiempo.

Si comparamos los puntos de correlación máxima de las distintas melodías generadas podremos ver cuales de ellas son más parecidas a los datos de entrenamiento y si alguna de ellas podría estar siendo generada fruto del sobre-aprendizaje.

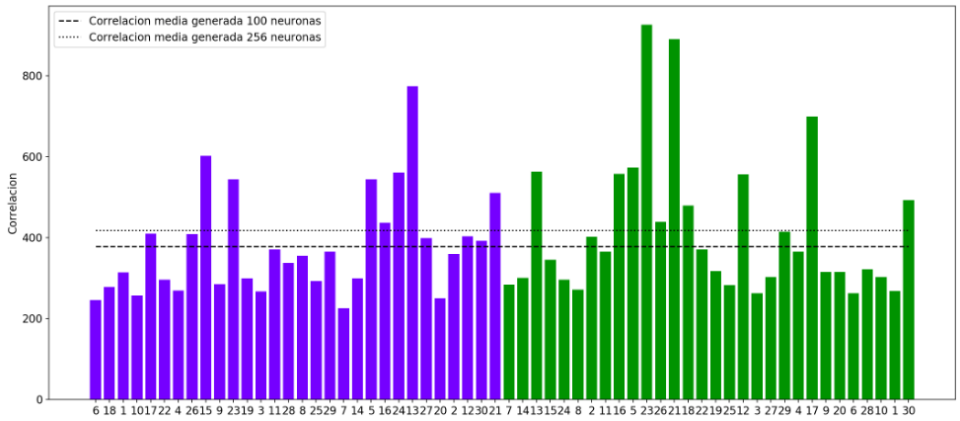

Figura 5.11: Correlación de las distintas melodías generadas para las melodías obtenidas de entrenar con 100 neuronas ocultas (azul) y con 256 neuronas ocultas (verde). La linea horizontal es la correlación media

Se puede observar en la figura 5.11 que las melodías generadas con el modelo de 256 neuronas ocultas parece tener un poquito más de sobre-aprendizaje que el de las 100 neuronas ocultas.

Estos resultados pese a que nos muestran que melodías de las generadas son un poco más parecidas que otras a los datos de entrenamiento no nos resuelven la pregunta de si son fruto de sobre-aprendizaje o no. Para esto necesitamos ponerlas en contexto y darles una comparativa.

En el caso de que dada una semilla la red generase una melodía igual o muy parecida a una de las de entrenamiento deberíamos obtener una correlación cruzada alta. Para simular este hecho calcularemos las correlaciones cruzadas de cada una de las melodías que conforman el conjunto de datos de entrenamiento y compararemos estos valores con las correlaciones cruzadas de los datos generados. Teniendo así una comparativa con melodías que se podrían generar en el caso de que la red tuviese mucho sobre-aprendizaje.

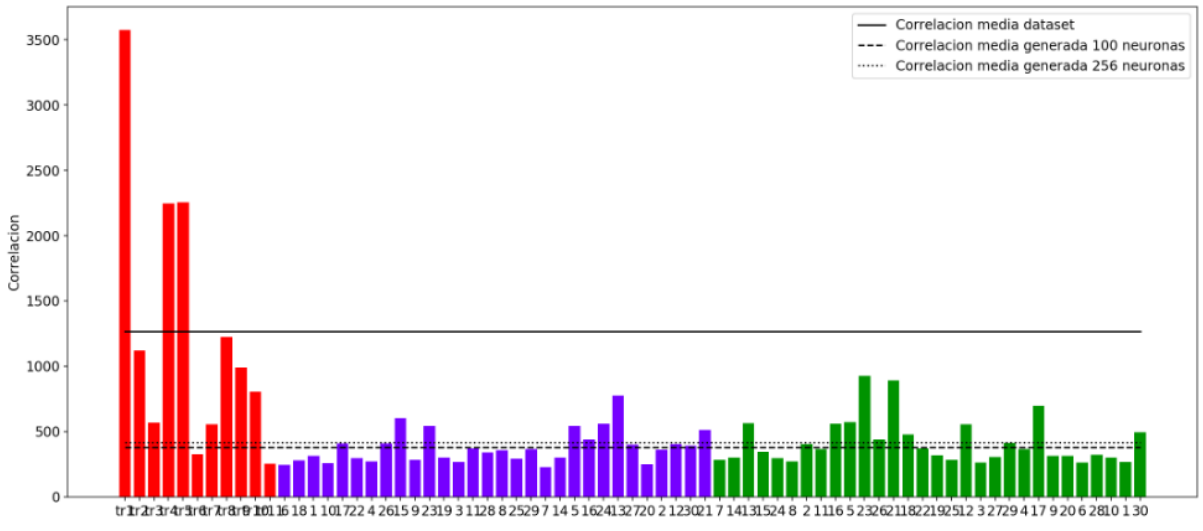

Figura 5.12: Comparación de las correlaciones de las melodías del conjunto de entrenamiento contra el fichero de entrenamiento (construido uniendo todas las sonatas) (rojo), con las correlaciones de las melodías generadas con 100 neuronas ocultas (azul) y generadas con 256 neuronas ocultas (verde).

Para visualizar el gráfico de la figura 5.12 con más facilidad se muestran dos lineas horizontales que se corresponden con la correlación cruzada media del conjunto de las melodías generadas (linea punteada) y del conjunto de las melodías de entrenamiento (linea continua). Podemos observar que en general las correlaciones de los ficheros generados están bastante por debajo de las correlaciones de los ficheros de entrenamiento. Aún así, en general, los modelos no tienen mucho sobre-aprendizaje.

Aún así, como se menciona en [37], *"siempre será bueno que tenga un poquito de sobre-aprendizaje para que pueda reproducir alguna de las combinaciones armónicas de notas/acordes"* que en este caso Chopin realiza.

5.3. Conclusiones.

Se puede observar, que la generación de modelos que resuelvan el modelado de series temporales es algo que está al alcance del usuario. Aunque exista la **limitación** de modelar problemas que requieren una alta capacidad de cómputo con equipos comunes, como se vio en los resultados de la sección 5.2.1 en general se pueden obtener resultados satisfactorios para problemas que no requieran tantos recursos como es el caso de la generación a partir de ficheros MIDI (ver sección 5.2.2).

Otra de las conclusiones extraídas de los resultados musicales es que es difícil realizar métricas que validen como de buena es la música sin necesidad de escucharla y pese a que las métricas utilizadas puede que no nos den información totalmente precisa de como de buenas son las melodías usadas si que nos dan una idea aproximada de que el modelo está consiguiendo generar datos, con una estructura aprendida de las melodías de entrenamiento.

Algo a realizar en el futuro sería encontrar métricas más precisas como podría ser, calcular la entropía a partir de la *escala fundamental* de la melodía siendo esta el conjunto de *símbolos fundamentales* formados por los conjuntos de notas que minimicen la entropía de la melodía[36]. El cálculo de estos símbolos es altamente no lineal por lo que para extraerlos se aplican algoritmos genéticos siendo esta una operación costosa. Esta solución nos dará un resultado más preciso de como de estructurada está la melodía.

Otra métrica posible, basada también en la entropía sería la de calcular la **entropía condicionada** hallando las probabilidades de un determinado símbolo dados N símbolos anteriores.

6

Conclusiones y trabajo futuro.

6.1. Conclusiones.

Pese a que hace años, cuando empezaron a implementarse modelos de redes recurrentes los resultados no eran muy buenos, hoy en día, gracias a las mejoras en los algoritmos de optimización, al desarrollo de los nuevos típos de redes recurrentes basadas en puertas y a la potencia que se puede conseguir de las GPUs estos modelos se han convertido en una poderosa herramienta.

La traducción, generación de música o texto, etiquetado de imágenes, creación de *chatbots*, son algunas de las fascinantes tareas que estos modelos han resuelto.

La implementación de modelos está al alcance de cualquier usuario, y de un modo bastante sencillo con librerías como Tensorflow o Keras.

A parte, como se ha podido mostrar con los resultados obtenidos cualquier usuario puede conseguir resultados interesantes con equipos normales. Pese a ello, también se ha podido ver que las capacidades de los equipos usados pueden limitar mucho el uso de modelos grandes con problemas complejos. Por ello el uso de las conocidas **clouds** como la **GoogleCloud** están a la orden del día.

6.2. Trabajo futuro.

El potencial de las redes recurrentes es enorme y es aplicable a diversos problemas. Dentro del campo de la generación musical, nos hemos quedado con ganas de obtener mejores resultados a partir de ficheros de audio en bruto. El trabajo futuro a realizar para obtener mejores resultados sería el de aplicar métodos de discretización de la onda que consigan extraer de una manera precisa la información relevante reduciendo así el coste de entrenamiento y mejorando también los resultados de las secuencias generadas.

Algo que vemos claro para todos los experimentos realizados es que disponer de una alta capacidad computacional habría permitido mejores resultados. Por ello, una de las actividades a llevar a cabo en el futuro sería la de obtener recursos suficientes que permitiesen realizar las siguientes actividades. Ampliar tanto las longitudes de las secuencias de entrenamiento como

los modelos usados, aumentar el número de datos de entrenamiento y mejorar las técnicas de traducción.

Estos modelos se podrían aplicar a la creación de traductores pequeños o *chatbots* y su implementación no sería muy distinta a la usada para este trabajo.

Bibliografía

[1] Cs231n: Convolutional neural networks for visual recognition. Abril 2017. http://cs231n.stanford.edu/.

[2] Jianqiang Ma. All of recurrent neural networks. *Medium*, 2016. https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[4] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):664–676, April 2017.

[5] Mark Gales. Statistical sequence modeling. *Cambridge University*.

[6] Laurene V. Fausett. *Fundamentals of Neural Networks*. Prentice Hall, us ed edition, 1993.

[7] Introduction to deep neural networks. *DL4J*, 2017. https://deeplearning4j.org/neuralnet-overview.

[8] Roger Grosse. Lecture 15: Exploding and vanishing gradients. *University of Toronto.*, 2017. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf.

[9] Erik Hallström. Backpropagation from the beggining. *Medium*, 2016. https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d.

[10] Andrew Hsu John McGonagle, George Shaikouski. Backpropagation. *Brilliant*, 2018. https://brilliant.org/wiki/backpropagation/.

[11] Suriyadeepan Ram. Unfolding rnns. *GitHub.io*, 2017. http://suriyadeepan.github.io/2017-01-07-unfolding-rnn/.

[12] Jeffrey L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[13] Understanding lstm networks. *Colah's blog*.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

[15] Jürgen Schmidhuber Felix A. Gers and Fred Cummins. Long short-term memory. 12:2451–2471, 10 2000.

[16] Simeon Kostadinov. Understanding gru networks. *Towards Data Science*, 2017. https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[17] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[18] Scikit-learn. About us. http://scikit-learn.org/stable/about.html.

[19] Cade Metz. Google just open sourced tensorflow, its artificial intelligence engine. *Wired*, 2015. https://www.wired.com/2015/11/google-open-sources-its-artificial-intelligence-engine/.

[20] Tensorflow. Tensorflow programmer's guide. https://www.tensorflow.org/programmers_guide/tensors, 2018.

[21] Cntk: The microsoft cognitive toolkit. https://docs.microsoft.com/en-us/cognitive-toolkit/. Accessed: 2018-06-12.

[22] Theano documentation. http://deeplearning.net/software/theano/. Accessed: 2018-06-12.

[23] Keras Documentation. Keras backends. https://keras.io/backend/. Accessed: 2018-05-12.

[24] Keras. Keras documentation. https://keras.io/. Accessed: 2018-05-12.

[25] Daniil Korbut. Machine learning translation and the google translate algorithm. *Stats and bots blog*, 2017. https://blog.statsbot.co/machine-learning-translation-96f0ed8f19e4.

[26] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[27] Ian Simon and Sageev Oore. Performance rnn: Generating music with expressive timing and dynamics. *Magenta Blog*, 2017. https://magenta.tensorflow.org/performance-rnn.

[28] Nipun Agarwala; Yuki Inoue and Axel Sly. Music composition using recurrent neural networks. *Stanford University*, 2017 (Accessed: 2018-05-10). https://web.stanford.edu/class/cs224n/reports/2762076.pdf.

[29] Roisin Loughran, Jacqueline Walker, Michael OÑeill, and Marion O'Farrell. The use of mel-frequency cepstral coefficients in musical instrument identification. 08 2008.

[30] Sebastian Ruder. On word embeddings. http://ruder.io/word-embeddings-1/, 2016. Accessed: 2018-05-21.

[31] Sujit Pal Antonio Gulli. *Deep Learning with Keras*. Packt Publishing Ltd., Livery Place, Birmingham, 2017.

[32] Music21: A toolkit for computer-aided musicology. http://web.mit.edu/music21/. Accessed: 2018-05-23.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, 2014.

[34] Normal order. http://openmusictheory.com/normalOrder.html. Accessed: 2018-06-14.

[35] Jaffe K. Febres G. Music viewed by its entropy content: A novel window for comparative analysis. 2017.

[36] Gerardo Febres and Klaus Jaffe. A fundamental scale of descriptions for analyzing information content of communication systems. *Entropy*, 17(4):1606–1633, 2015.

[37] Yuki Inoue; Nipun Agarwala and Alex Sly. Music composition using recurrent neural networks. technical report. *Institute of Computer Science, University of Tartu*, 2017.

Anexos

A

Código

A.1. Código para la generación de texto.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Código A.1: Dependencias.

```
class DataManager:
def __init__(self, input_file, batch_size=1):
self.input_file = input_file
self.batch_size = batch_size

# Read data file:
self.data = open(self.input_file, 'r').read()
self.data_size = len(self.data)
print 'Initial data has %d characters.' %self.data_size

# Adjust data to integer number of batches:
self.num_batches = self.data_size/self.batch_size
self.data = self.data[:self.batch_size*self.num_batches]
print 'batch_size = %d, num_batches = %d' %(self.batch_size, self.num_batches)

# Get vocabulary (set of chars):
self.chars = list(set(self.data))
self.data_size = len(self.data)
self.vocab_size = len(self.chars)
print 'Final data has %d characters, %d unique.' %(self.data_size, self.
    vocab_size)

# Dictionaries to map from char to index and vice-versa:
self.char_to_ix = { ch:i for i,ch in enumerate(self.chars) }
self.ix_to_char = { i:ch for i,ch in enumerate(self.chars) }

# Data in numerical format and organized in batches:
self.data_num = np.array([self.char_to_ix[ch] for ch in self.data])
self.data_num = self.data_num.reshape((self.batch_size, self.num_batches))
print "Numerical data organized as matrix with shape " + str(self.data_num.shape)
    + ", one batch per column"
```

```
30
31 # Initialize batch counter:
32 self.next_batch_counter = 0
33
34 def next_sequential_batch(self, n_steps):
35 # Check, num_batches > n_steps:
36 if n_steps >= self.num_batches:
37 print "In DataManager.next_sequential_batch():"
38 print "   Requested %d batches, but there are only %d batches available..." %(
       n_steps, self.num_batches)
39 return None
40
41 # Find next batch with n_steps, need n_steps+1 because we need the next char as
       target:
42 ix = self.next_batch_counter
43 self.next_batch_counter += n_steps
44 if self.next_batch_counter >= self.num_batches:
45 ix = 0
46 self.next_batch_counter = n_steps
47
48 # Return batch, both input and output (target):
49 return self.data_num[:,ix:self.next_batch_counter], self.data_num[:,(ix+1):(self.
       next_batch_counter+1)]
50
51 def reset_batch_counter(self):
52 # Resets the bacth counter to 0
53 self.next_batch_counter = 0
54
55 def next_random_batch(self, n_steps):
56 # Check, num_batches > n_steps:
57 if n_steps >= self.num_batches:
58 print "In DataManager.next_random_batch():"
59 print "   Requested %d batches, but there are only %d batches available..." %(
       n_steps, self.num_batches)
60 return None
61
62 # Get and return random batch:
63 ix = np.random.randint(0, self.num_batches-n_steps)
64 return self.data_num[:,ix:(ix+n_steps)], self.data_num[:,(ix+1):(ix+n_steps+1)]
65
66 def next_sequential_batch_one_hot(self, n_steps):
67 # Returns the next batch with one-hot codification
68 x, y = self.next_sequential_batch(n_steps)
69 x1h = 1*(x[:,:,None] == np.arange(self.vocab_size).reshape((1, 1, self.vocab_size
       )))
70 y1h = 1*(y[:,:,None] == np.arange(self.vocab_size).reshape((1, 1, self.vocab_size
       )))
71 return  x1h, y1h
```

Código A.2: Clase para la gestión de los datos.

```
1 # Nuevo grafo:
2 tf.reset_default_graph()
3
4 # Placeholders entrada y salida:
5 x = tf.placeholder(tf.float32, [batch_size, n_steps, vocab_size], name="x")
6 y = tf.placeholder(tf.float32, [batch_size, n_steps, vocab_size], name="y")
7
8 # Placeholders para el estado inicial:
9 initial_c = tf.placeholder(tf.float32, [batch_size, n_hidden], name="c")
10 initial_h = tf.placeholder(tf.float32, [batch_size, n_hidden], name="h")
11 initial_state = tf.contrib.rnn.LSTMStateTuple(initial_c, initial_h)
12
13 # Capa LSTM:
```

```
cell = tf.contrib.rnn.LSTMCell(n_hidden)
outputs, state = tf.nn.dynamic_rnn(cell, x, initial_state=initial_state, dtype=tf.float32)

# Reshape de la salida de LSTM:
reshaped_outputs = tf.reshape(outputs, [-1, n_hidden])

# Capa softmax:
W = tf.Variable(tf.truncated_normal([n_hidden, vocab_size]), name="W")
b = tf.Variable(tf.constant(0.1, shape=[vocab_size]), name="b")
z = tf.matmul(reshaped_outputs, W) + b

# Loss:
y_hat = tf.reshape(z, [-1, n_steps, vocab_size], name="y_hat")

loss = tf.contrib.seq2seq.sequence_loss(y_hat,
tf.argmax(y,2),
tf.ones([batch_size, n_steps], dtype=tf.float32),
average_across_timesteps=True,
average_across_batch=True)

# Summary para tensorboard (lo guardo antes de introducir el optimizador para no complicar el grafo):
writer = tf.summary.FileWriter("./logs", tf.get_default_graph())
writer.close()

# Optimizador:
train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)
# Objeto Saver para salvar y cargar datos:
saver = tf.train.Saver()
```

Código A.3: Creación del modelo en Tensorflow.

```
from datetime import datetime
start_time = datetime.now()
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
costs = []
data.reset_batch_counter()
i = 0
while i<500000:
# Get next batch:
mbx, mby = data.next_sequential_batch_one_hot(n_steps)

# Whenever the sequence starts again we reset the network state:
if data.next_batch_counter == n_steps:
current_c = np.zeros((batch_size, n_hidden))
current_h = np.zeros((batch_size, n_hidden))
print "--- reset state ---"

# Perform weight update:
_, cstate = sess.run([train_step, state], {x: mbx, y: mby, initial_c: current_c, initial_h: current_h})

# Print loss every 1000 iterations:
if i%1000 == 0:
c = sess.run(loss, {x: mbx, y: mby, initial_c: current_c, initial_h: current_h})
if i==0:
smooth_loss = c
else:
smooth_loss = 0.99*smooth_loss + 0.01*c
costs.append(smooth_loss)
print "step %d, cost %g" %(i, smooth_loss)

```

```
# Save the network every 10000 iterations:
if i%10000 == 0:
fout = "../output/quijote_%07d.ckpt" % i
saver.save(sess, fout)

# Update network state:
current_c = cstate[0]
current_h = cstate[1]

# Increase iteration counter:
i += 1

time_elapsed = datetime.now() - start_time
print('Time elapsed (hh:mm:ss.ms) {}'.format(time_elapsed))

plt.plot(costs)
plt.xlabel("Epoch")
plt.ylabel("Cost")
plt.show()
```

Código A.4: Etapa de entrenamiento durante 500.000 pasos de tiempo.

```
ifile = 490000

# Nuevo grafo:
tf.reset_default_graph()

# Placeholder entrada, solo 1 step:
x = tf.placeholder(tf.float32, [batch_size, 1, vocab_size], name="x")

# Placeholder para el estado inicial:
initial_c = tf.placeholder(tf.float32, [batch_size, n_hidden])
initial_h = tf.placeholder(tf.float32, [batch_size, n_hidden])
initial_state = tf.contrib.rnn.LSTMStateTuple(initial_c, initial_h)

# Capa LSTM:
cell = tf.contrib.rnn.LSTMCell(n_hidden)
outputs, state = tf.nn.dynamic_rnn(cell, x, initial_state=initial_state, dtype=tf
    .float32)

# Reshape de la salida de LSTM:
reshaped_outputs = tf.reshape(outputs, [-1, n_hidden])

# Capa softmax:
W = tf.Variable(tf.truncated_normal([n_hidden, vocab_size]), name="W")
b = tf.Variable(tf.constant(0.1, shape=[vocab_size]), name="b")
z = tf.matmul(reshaped_outputs, W) + b

prediction = tf.nn.softmax(z)

# Summary para tensorboard:
writer = tf.summary.FileWriter("./logs", tf.get_default_graph())
writer.close()

# Objeto Saver para salvar y cargar datos:
saver = tf.train.Saver()

# Creo sesion y cargo el modelo:
sess = tf.InteractiveSession()
fin = "../output/fork_%07d.ckpt" % ifile
saver.restore(sess, fin)

# Generacion de texto:
init_char = 's'
```

```
semilla = np.array([[char_to_ix[init_char]]])
print semilla
semilla = 1*(semilla[:,:,None] == np.arange(vocab_size).reshape((1, 1, vocab_size)))
print semilla
current_c = np.zeros((batch_size, n_hidden))
current_h = np.zeros((batch_size, n_hidden))
cadena = init_char
for i in range(5000):

probs, cstate = sess.run([prediction, state], {x: semilla, initial_c: current_c, initial_h: current_h})
next_char = np.random.choice(range(vocab_size), p=probs.ravel())
#next_char = np.argmax(probs.ravel())

cadena += ix_to_char[next_char]

semilla = np.array([[next_char]])
semilla = 1*(semilla[:,:,None] == np.arange(vocab_size).reshape((1, 1, vocab_size)))
current_c = cstate[0]
current_h = cstate[1]

print
print cadena
```

Código A.5: Código para la generación de texto a partir de lo aprendido en el entrenamiento.

A.2. Código para la generación de música a partir de WAV.

```
import os
import scipy.io.wavfile as wf
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.layers import Dense, TimeDistributed, InputLayer, Dropout, Activation
from keras.layers.recurrent import LSTM
from IPython.display import Audio
from pipes import quote
```

Código A.6: Dependencias.

```
def wav_to_np(file):
  wav_file_data = wf.read(file)
  # Extraemos el sample rate (numero de valores por segundo)
  rate = wav_file_data[0]
  values = wav_file_data[1]
  # En el caso de que haya dos canales elegimos solo uno de los dos
  if len(values.shape) == 2:
    data = values[:,0]
  else:
    data = values
  norm_data = data/300
  norm_data = norm_data.astype('int16')
  norm_data = remove_zeros(norm_data)
  return data, norm_data, rate

def np_to_wav(filename, rate, data):
  # Revertimos la normalizacion
  data *=300
  data = data.astype('int16')
```

```
20   wf.write(filename, rate, data)
```

Código A.7: Función de conversión de WAV a numpy y viceversa.

```
1  def to_dict(data):
2    vals = set(data)
3    direct_dict = {i:e for i,e in enumerate(vals)}
4    inverse_dict = {e:i for i, e in enumerate(vals)}
5    return direct_dict, inverse_dict
6
7  def to_one_hot(data, inverse_dict):
8    mapped_data_values = list(map(lambda x: inverse_dict[x], data))
9    max_data = max(inverse_dict.values())+1
10   return np.eye(max_data)[mapped_data_values]
11
12 def one_hot_to_val(data, direct_dict):
13   result_data = []
14   for e in data:
15     argmax = e.argmax()
16     result_data.append(direct_dict[argmax])
17   return np.array(result_data)
```

Código A.8: Funciones auxiliares.

```
1  def to_dict(data):
2    vals = set(data)
3    direct_dict = {i:e for i,e in enumerate(vals)}
4    inverse_dict = {e:i for i, e in enumerate(vals)}
5    return direct_dict, inverse_dict
6
7  def to_one_hot(data, inverse_dict):
8    mapped_data_values = list(map(lambda x: inverse_dict[x], data))
9    max_data = max(inverse_dict.values())+1
10   return np.eye(max_data)[mapped_data_values]
11
12 def one_hot_to_val(data, direct_dict):
13   result_data = []
14   for e in data:
15     argmax = e.argmax()
16     result_data.append(direct_dict[argmax])
17   return np.array(result_data)
18
19 def get_next_block(data, size, pos):
20   return data[pos:pos+size], data[pos+1:pos+1+size]
21
22 def join_blocks(blocks):
23   return np.concatenate(blocks)
```

Código A.9: Funciones auxiliares.

```
1  class DataGen(keras.utils.Sequence):
2    def __init__(self, data, seq_length, inverse_dict):
3      self.pos = 0
4      self.data = data
5      self.seq_length = seq_length
6      self.inverse_dict = inverse_dict
7      self.num_batches = 1
8
9    def __len__(self):
10     return self.num_batches
11
12   def __getitem__(self, index):
13     X = []
14     y = []
```

```
    batch_size = int((len(self.data)-seq_length+1)/self.num_batches)
    self.pos = 0
    for _ in range(batch_size):
      data_X, data_y = self.gen_values(self.pos)
      X.append(data_X)
      y.append(data_y)
      self.pos+=1

    return np.array(X), np.array(y)

  def gen_values(self, pos):
    data_X, data_y = get_next_block(self.data, self.seq_length, pos)
    data_X = to_one_hot(data_X, self.inverse_dict)
    data_y = to_one_hot(data_y, self.inverse_dict)
    return data_X, data_y
```

Código A.10: Clase generadora de los datos.

```
filename = 'data/violin.wav'
dat, data, rate = wav_to_np(filename)
direct_dict, inverse_dict = to_dict(data)
one_hot_x = to_one_hot(dataX, inverse_dict)
ini_pos = 0
unique_values_length = len(direct_dict)
seq_length = 100
total_seq = len(data)
```

Código A.11: Lectura de datos e inicialización de hiperparámetros.

```
model = Sequential([
  LSTM(units=seq_length, input_shape=(None , unique_values_length),
    return_sequences=True),
  TimeDistributed(Dense(units=unique_values_length), input_shape=(seq_length,)),
  Activation("softmax")
])
model.compile(loss='categorical_crossentropy', optimizer='adam')

training_generator = DataGen(data, seq_length, inverse_dict)

model.fit_generator(generator=training_generator, epochs=30, verbose=True)
```

Código A.12: Definición del modelo y ejecución del mismo.

```
# Cogemos el primer trozo como semilla
test_generator = DataGen(data, seq_length, inverse_dict)
seed_seq, y = test_generator.gen_values(0)
# Reshape.
seed_seq = np.reshape(seed_seq, (1, seed_seq.shape[0], seed_seq.shape[1]))
# Guardamos la secuencia en output.
output = None
for it in range(100):
  print("\r{}".format(it), end="")
  # Generates new value
  seedSeqNew = model.predict(seed_seq)
  newSeq = seedSeqNew[0].copy()
  if output is None:
    output = newSeq
  else:
    output = np.concatenate((output, newSeq))
  newSeq = np.reshape(newSeq, (1, newSeq.shape[0], newSeq.shape[1]))
  seed_seq = newSeq
output=np.array(output)
output = one_hot_to_val(output, direct_dict)

```

```
np_to_wav("audio.wav", rate, output.astype(np.int16))
Audio("audio.wav")
```

Código A.13: Código de generación de música.

A.3. Código para la generación de música a partir de MIDI.

```
import os, glob
import numpy as np
import matplotlib.pyplot as plt

from __future__ import print_function

import keras
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, TimeDistributed, InputLayer, Dropout, Activation
from keras.layers.recurrent import LSTM, GRU

from music21 import converter, instrument, note, chord, duration, stream,
    environment
```

Código A.14: Dependencias.

```
class DataManager(keras.utils.Sequence):
  def __init__(self, file_, seq_length, filename=None, batch_size=1):
    self.durations = set()
    self.data = self.extract_notes_file(file_, filename)
    self.data_size = len(self.data)
    print("Los datos iniciales tienen {} valores".format(self.data_size))

    self.seq_length = seq_length
    self.batch_size = batch_size
    self.num_batches = int((self.data_size-(seq_length-1))/self.batch_size)

    #Ajustamos los datos al numero de batches
    #self.data = self.data[:(self.batch_size*self.num_batches+seq_length)]
    print("\nDimension de batch = {}. Numero de batches = {}".format(self.
    batch_size, self.num_batches))

    # Extraemos el numero distinto de notas y duraciones de las mismas
    self.notes = set(self.data)

    # Dado que el valor a predecir sera una concatenacion entre la nota y la
    duracion, la dimension del "vocabulario"
    # sera la suma de ambas longitudes.
    self.vocab_size = len(self.notes)
    self.data_size = len(self.data)

    print("Los datos finales tienen {} valores".format(self.data_size))
    print("Los datos cuentan con {} valores distintas".format(self.vocab_size))

    # Diccionarios para mapear notas y duraciones a indices
    self.notes_to_ix = { ch:i for i,ch in enumerate(self.notes) }
    self.ix_to_notes = { i:ch for i,ch in enumerate(self.notes) }

    self.notes_num =[self.notes_to_ix[ch] for ch in self.data]
    self.data_num = np.array(self.notes_num)
    self.data_one_hot = self.data_to_one_hot(self.data_num)
    self.current_idx = 0
    self.idx = []
    self.batch_pos = 0
    #self.save_batches(self.seq_length)
```

```
38
39  def extract_notes_file(self, dir_, filename=None):
40    self.timeSignature = 0
41    notes = []
42    files = glob.glob(dir_+"/*")
43    tot = len(files)
44    i = 1
45    for f in files:
46      if filename is not None:
47        f = dir_+"/"+filename
48      print("\r{}\{}".format(i, tot), end="")
49      i+=1
50      midi = converter.parse(f)
51      notes_to_parse = None
52      parts = instrument.partitionByInstrument(midi)
53      if parts: # file has instrument parts
54        notes_to_parse = parts.parts[0].recurse()
55      else: # file has notes in a flat structure
56        notes_to_parse = midi.flat.notes
57      for element in notes_to_parse:
58        if element.duration < duration.convertTypeToQuarterLength("2048th"):
59          duration_ = "2048th"
60        else:
61          duration_ = element.duration.type
62        if duration_ != "inexpressible" and duration_ != "complex":
63          self.durations.add(duration_)
64        if isinstance(element, note.Note):
65          notes.append(str(element.pitch))
66        elif isinstance(element, chord.Chord):
67          notes.append('.'.join(str(n) for n in element.normalOrder))
68        if isinstance(element, note.Note) or isinstance(element, chord.Chord):
69          if duration_ != "inexpressible" and duration_ != "complex":
70            notes.append(duration_)
71          else:
72              notes.append("quarter")
73      if filename is not None:
74        break
75    return notes
76
77  def create_midi_file(self, prediction, filename):
78    output_notes = []
79    offset = 0.0
80    # Crear la secuencia con notas, acordes y duraciones
81    i=0
82    while i < len(prediction):
83      print("\rEscribiendo: {}\{}".format(i, len(prediction)), end="")
84      e = prediction[i]
85      if i+1 >= len(prediction):
86        break
87      if prediction[i+1] in self.durations:
88        durationValue = duration.convertTypeToQuarterLength(prediction[i+1])
89        i+=1
90      else:
91        durationValue = 0.5
92      if ('.' in e) or e.isdigit():
93        notes_in_chord = e.split('.')
94        notes = []
95        for current_note in notes_in_chord:
96          new_note = note.Note(int(current_note))
97          new_note.storedInstrument = instrument.Piano()
98          notes.append(new_note)
99        new_chord = chord.Chord(notes)
100       new_chord.quarterLength = durationValue
```

```
        new_chord.offset = offset
        output_notes.append(new_chord)
        offset += durationValue
      elif e not in self.durations:
        new_note = note.Note(e)
        new_note.offset = offset
        new_note.quarterLength = durationValue
        new_note.storedInstrument = instrument.Piano()
        output_notes.append(new_note)
        # Aumentamos el offset acorde a la duracion
        offset += durationValue
      i+=1
    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp=filename)
    return midi_stream

  def data_to_one_hot(self, data):
    one_hot_data = []
    for note in data:
      note_one_hot = self.value_to_one_hot(note, len(self.notes))
      one_hot_data.append(note_one_hot)
    return np.array(one_hot_data)

  def value_to_one_hot(self, value, data_size):
    aux = np.zeros(data_size)
    np.put(aux, value, 1)
    return aux.astype(np.uint8)

  def generate_training_values(self, seq_length):
    dataX = []
    dataY = []
    for pos in range(len(self.data_one_hot) - seq_length-1):
      dataX.append(self.data_one_hot[pos:pos+seq_length])
      dataY.append(self.data_one_hot[pos+1:pos+seq_length+1])
    return np.array(dataX), np.array(dataY)

  def one_hot_value_to_notes(self, value):
    note_one_hot = value
    note = np.random.choice(np.arange(len(note_one_hot)), p=note_one_hot)
    #note = np.argmax(note_one_hot)

    return self.ix_to_notes[note]
```

Código A.15: Clase para la gestión de los datos.

```
filepath = "drive/midi_music/backup/weights.{epoch:02d}.hdf5"
save_weights = keras.callbacks.ModelCheckpoint(filepath,
                             verbose=0,
                             save_best_only=False,
                             save_weights_only=False,
                             mode='auto', period=5)
model = Sequential([
  LSTM(units=256, input_shape=(None, data.vocab_size), return_sequences=True),
  Dropout(0.3),
  TimeDistributed(Dense(units=data.vocab_size), input_shape=(data.seq_length,)),
  Activation("softmax")
])
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Código A.16: Creación del modelo en Keras.

```
history = model.fit(X, y, epochs=300, verbose=True, callbacks=[save_weights])
```

Código A.17: Etapa de entrenamiento durante 300 épocas.

```
1
2  index = 10000 #Indice de donde cogeremos la semilla
3  note_one_hot = data.value_to_one_hot(data.data_num[index], len(data.notes))
4  duration_one_hot = data.value_to_one_hot(data.data_num[index+1], len(data.notes))
5
6  # Semilla en one hot
7  note_one_hot = np.reshape(note_one_hot, (1, 1, note_one_hot.shape[0]))
8  duration_one_hot = np.reshape(duration_one_hot, (1, 1, duration_one_hot.shape[0])
       )
9  seed_one_hot = np.concatenate((note_one_hot, duration_one_hot), axis=1)
10
11 generated = []
12 generated.append(data.one_hot_value_to_notes(note_one_hot.flatten()))
13 generated.append(data.one_hot_value_to_notes(duration_one_hot.flatten()))
14
15 song_size = 500
16
17 for i in range(song_size):
18   print("\r{}\{}".format(i, song_size), end = "")
19   prediction = model.predict(seed_one_hot)[0][-1]
20   prediction = np.reshape(prediction, (1, 1, prediction.shape[0]))
21   seed_one_hot = np.concatenate((seed_one_hot, prediction), axis=1)
22
23   value = data.one_hot_value_to_notes(prediction.flatten())
24   generated.append(value)
25
26 data.create_midi_file(generated, "drive/midi_music/output/classic_little_{}_{}
       _epoch.mid".format(count, pesos))
```

Código A.18: Código para la generación de música a partir de lo aprendido en el entrenamiento.

B

Otros resultados.

B.1. Código linux generado.

```
#ifdef CONFIG_FUTEX
        p->parent_exec *stack = task_struct_cachep;

static inline void page_cache(struct task_struct *task)
{
        struct task_struct *task = current->files;
                        /*
                 * The seccomp threads
                        if (new_fd) {
                                clone_flags | CLONE_VM)
                unshare_flags |= CLONE_THREAD
                free_task(tsk);
        return 0;
}

static inline void free_task_struct(struct task_struct *tsk)
{
        struct task_struct *task = current->files;
                        /*
                 * The seccomp threads* shoun binfmt handlers to the
         * free the new user namespace
         * from the new as the new process tree
 * but lote for filiia.  Cov.h>
#include <linux/compiler.h>
#include <linux/kthread.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compiler.h>
#include <linux/compile which visitor, *new_fd,
                                                        new_fd = NULL;
                        ttatall = copy_fs(clone_flags, parent_struct_cachep, tsk);
        p->pid_t.h>
```

Figura B.1: Código generado a partir del kernel de linux.